Marco André Binz Hennes

PLATAFORMA PARA TESTES DE CIRCUITOS DIGITAIS

Dissertação apresentada ao Programa de Pós-Graduação em Sistemas e Processos Industriais - Mestrado, Área de concentração em Controle e Otimização de Processos Industriais, Universidade de Santa Cruz do Sul - UNISC, como requisito parcial para obtenção do título de Mestre em Sistemas e Processos Industriais.

Orientador: Prof. Dr. Rafael Ramos dos Santos

Santa Cruz do Sul, maio de 2009

Agradecimentos

Aos meus pais (Walter e Marion), irmãos (Carlos, Fernando e Roberto) e avós (Heinrich, Beatriz, Arno e Holanda) pelo apoio à esse trabalho.
À UNISC, pelo apoio para a realização desse trabalho.
À CAPES, pelo apoio financeiro com a manutenção da bolsa de auxílio.
A minha namorada Marcelli.
Aos professores Rafael Ramos dos Santos e Rolf Fredi Molz .
Aos colegas da turma do GPSEM (Rafael, Rolf, Lucio, Joel, Adriano, Jorge, Vitor, Felipe, Robert, Felipe, Battisti, Ricardo, Maicon), pelas sugestões recebidas.
Agradeço aos Professores Doutores do Programa de Pós-graduação em Sistemas e Processos Industriais (João Carlos Furtado, Marco Flôres Ferrão, Rafael Ramos dos Santos, Rejane Frozza, Rolf Fredi Molz, Ruben Edgardo Panta Pazos, Liane Mählmann Kipper, Geraldo L. Crossetti) e à secretária Janaina Ramires Haas .
Deixo meu agradecimento ao médico Paulo Cezar Schutz.
Ao $LaTeX$ *pela formatação desse trabalho.*
E finalmente agradeçao ao **chimarrão** *, que me proporcionou a paciência e foi meu companheiro durante longas noites de escrita desse trabalho.*

"A genialidade é 1% inspiração e 99% transpiração."(Thomas Edison)
"Dar menos que o seu melhor é sacrificar o dom que recebeu."(Steve Prefontaine)
"Reparta o seu conhecimento. É uma forma de alcançar a imortalidade."(Dalai Lama)
"Vencer a si próprio é a maior das vitórias."(Platão)

RESUMO

Com o avanço da tecnologia de produção de circuitos integrados, é necessário um maior esforço para concepção e verificação destes circuitos. O teste de circuitos integrados envolve diversas etapas e seu objetivo é identificar e isolar dispositivos falhos. Quando um novo circuito integrado é projetado, e antes que seja enviado para produção, a primeira etapa deve verificar se o projeto está correto. Com o crescimento da microeletrônica no país, com certeza será necessário que sejam criados dispositivos de testes cada vez mais eficientes e de preferência com custos acessíveis às empresas de menor porte. O presente trabalho visa a criação de um ambiente de verificação de baixo custo usando uma linguagem de programação que permita verificar os circuitos digitais. Esse trabalho descreve uma plataforma de verificação de circuitos integrados que emprega testes estruturais. Essa análise é feita em circuitos combinacionais do *benchmark* ISCAS85 (HANSEN et al. 1999) para falhas do tipo *stuck-at*. A plataforma de verificação foi desenvolvida com componentes de baixo custo, o que dá à plataforma um grande diferencial. Além disso, o ambiente realiza tanto verificação em software, como em hardware. Para a verificação realizada em hardware, o ambiente emula o circuito em um FPGA com o propósito de comprovar a simulação em um ambiente real. Outro diferencial importante é a integração dos ambientes de simulação, permitindo realizar verificação em software, como em hardware.

Palavras-chave: Simulação de falhas, emulação de falhas, teste de circuitos integrados, verificação, falhas *stuck-at* .

ABSTRACT

With the advance of technology for integrated circuits production, it is necessary a greater effort to design and verify these circuits. The test of integrated circuits involves several steps and its goal is to identify and isolate faulty devices. When a new integrated circuit is designed, and before it is sent to production, the first step is to verify that the project is correct. With the growth of microelectronics in the country, certainly it will be needed to create new efficient testing devices and preferably with affordable cost to small sized companies. This research aims at creating an environment of low cost tests using a programming language, which allows testing in digital circuits. This paper describes a platform for testing integrated circuits by using structural testing. This analysis is done in combinational circuits of ISCAS85 benchmark (HANSEN et al. 1999) for failures of type stuck-at. The testing platform was developed with low cost components, giving the platform a large differential. Moreover, the environmental performed both tests, in software and hardware. For tests with hardware, the environment emulates the circuit in a FPGA with the aim to prove the simulated tests in in a real environment. Another important difference is the integration of the environments of simulation, allowing testing in software as well as in hardware.

Keywords: *fault simulation, fault emulation, test of integrated circuits, verification, stuck-at fault.*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ARM	Advanced RISC Machine
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
CAD	Computer-Aided Design
DFT	Design for Testability
DUT	Device Under Test
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
MMU	Memory Management Unit
NFS	Network File System
PI	Primary Input
PO	Primary Output
PODEM	Path-Oriented Decision Making
ROBDD	Reduced Ordered Binary Decision Diagram
SOC	System-on-a-Chip
s-a-0	Stuck-At-0
s-a-1	Stuck-At-1
SSA	Single Stuck-At
SSBDD	Structurally Synthesized Binary Decision Diagram
VLSI	Very Large Scale Integrated Circuits

SUMÁRIO

1 INTRODUÇÃO

Analisando de uma forma histórica, a indústria de microeletrônica[1] no Brasil mostra que a fabricação de componentes semicondutores[2] já teve uma participação mais expressiva no mercado nacional.Por isso o Brasil constatou, a partir de 2003, quando o governo começou a incentivar a microeletrônica no país, que o segmento de microeletrônica cresce rapidamente a cada dia e seriam necessários investimentos maiores nessa área. A área da microeletrônica engloba tanto os processos físico-químicos de fabricação dos circuitos integrados[3], como o projeto do circuito em si.

1.1 Motivação e Objetivos

Com o avanço da tecnologia de produção de circuitos integrados, é necessário um maior esforço para a concepção e verificação destes circuitos. Esse aumento implica em um crescimento do tempo e do custo do projeto. Por outro lado, o grande avanço tecnológico faz com que o ciclo de vida de um produto eletrônico seja a cada dia menor e portanto o *time-to-market* se torna mais importante. Além disso, com o aumento da complexidade, a concepção, verificação e teste se torna mais difícil. Entretanto a complexividade dos circuitos também cresce, o que dificulta a concepção e a verificação de um novo circuito integrado. De um modo geral, pode-se dividir o projeto de um circuito integrado em dois níveis: o nível de projeto e o de fabricação, conforme mostrado na Figura 1. O nível de projeto engloba toda a descrição do hardware numa linguagem apropriada e o uso de ferramentas para a simulação e síntese. Já o nível de fabricação envolve desde a fabricação propriamente dita do circuito integrado até a parte de testes físicos e

[1] A microeletrônica é um ramo da eletrônica, voltado à integração de circuitos eletrônicos, promovendo uma miniaturização dos componentes em escala microscópica.

[2] Sólidos cristalinos de condutividade elétrica intermediária entre condutores e isolantes. Seu emprego é importante na fabricação de componentes eletrônicos(diodos, transistores, microprocessadores).

[3] Um circuito integrado, também conhecido por *chip*, é um dispositivo microeletrônico que consiste de muitos transístores e outros componentes interligados capazes de desempenhar uma ou mais funções.

lógicos. O teste deve verificar se o circuito está de acordo com o projeto, se o circuito apresenta um comportamento lógico correto (teste lógico) e se a implementação física do circuito espelha seu esquemático[4] (teste físico).

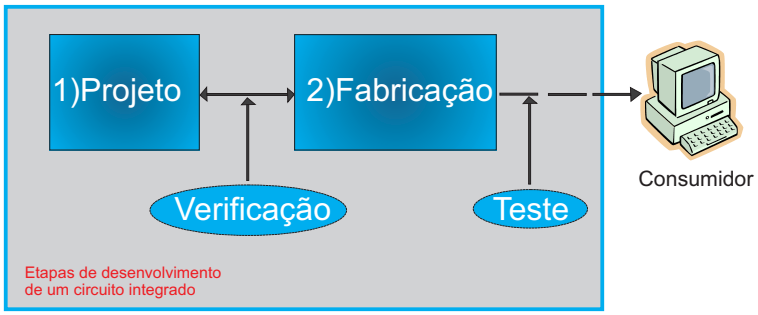


Figura 1: Processo de fabricação de um circuito integrado

A parte de testes é fundamental para se ter certeza do funcionamento correto do circuito. Durante o processo de produção de um circuito integrado, ele é submetido a diferentes tipos de testes. Um dos testes mais utilizados é o teste estrutural, baseado em falhas, o qual permite a obtenção de uma medida quantitativa da efetividade do teste (cobertura de falhas) (LUBASZEWSKI M.; COTA 2002). Qualquer método de teste é realizado através do uso de estímulos de entrada seguido pela observação dos valores das saídas do circuito, conforme mostrado na Figura 2. Esses estímulos na entrada são chamados de padrões de teste e , na saída, de resultados de testes. Com esses padrões de testes é avaliada a cobertura de falhas, que é o percentual de falhas do circuito que conseguem ser detectadas. A cobertura de falhas depende da testabilidade do circuito, que é mensurada pela facilidade ou pela dificuldade de controle e observação dos diversos pontos de falha.

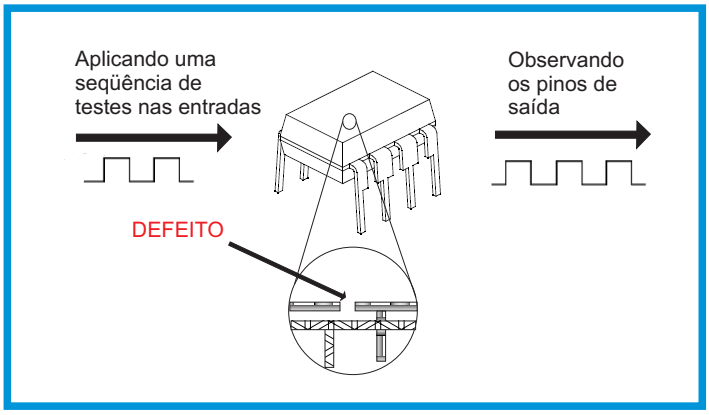


Figura 2: Teste de um ciruito integrado

[4]É uma representação simplificada de um circuito eletrônico.

O teste de circuitos integrados envolve diversas etapas e seu objetivo é identificar e isolar dispositivos falhos. Quando um novo circuito integrado é projetado, e antes que seja enviado para produção, a primeira etapa deve verificar se o projeto está correto. Se houver algum erro e não for identificado na etapa de verificação, serão desperdiçados recursos financeiros e humanos. Então, com o crescimento da microeletrônica no país com certeza será necessário que sejam criados dispositivos de verificação cada vez mais eficientes e de preferência com custos acessíveis às empresas de menor porte.

O presente trabalho visa a criação de uma plataforma de verificação de baixo custo. A plataforma proposta foi usada para verificação de circuitos digitais. Essa plataforma proposta foi implementada em Java e usa mais duas ferramentas EDA (*Electronic Design Automation*) para se verificar os circuitos. É possível dividir essa plataforma em verificação em software e em hardware. Na verificação em software, é possível com uma descrição de um circuito, usar algoritmos para se testar esse circuito. Já na verificação em hardware, foi desenvolvida uma plataforma de baixo custo que usa um microcontrolador e um FPGA (*Field Programmable Gate Array*). Tanto a verificação em software, como a em hardware, tem o propósito de se conseguir um circuito que funcione corretamente. Nos dois casos são aplicados vetores de teste na entrada do circuito e são analisadas suas saídas. Na verificação em hardware, o ambiente emula o circuito em um FPGA com o propósito de se comprovar os testes em simulação em um ambiente real.

1.2 Contribuições

As principais contribuições deste trabalho são:

- Proposta de uma metodologia de testes estruturais para descrições em HDL[5] (*Hardware Description Language*)
- Projeto de um testador de baixo custo usando uma linguagem de programação para a aplicação em simulação e emulação de circuitos em hardware
- Uso de um microcontrolador novo, que possui um sistema operacional[6], para verificação de circuitos integrados

[5]É qualquer linguagem dentro de um grupo de linguagens da computação para a descrição formal de circuitos eletrônicos.

[6]É um programa ou um conjunto de programas cuja função é servir de interface entre um computador e o usuário.

- Emprego de FPGAs para emulação do circuito, estimulação e observação do comportamento
- Plataforma de verificação flexível

1.3 Organização do trabalho

O conteúdo deste trabalho é organizado como segue. No capítulo 1 apresenta-se o mercado de semicondutores no Brasil e a necessidade de se realizar testes quando se cria um novo circuito digital. O capítulo 2 apresenta algumas definições importantes na área de testes de sistemas digitais e introduz a metodologia empregada. O capítulo 3 mostra os tipos de algoritmos usados em testes e os modelos de representação. Já o capítulo 4, fala sobre a arquitetura de verificação proposta e apresenta a ferramenta de verificação desenvolvida. O capítulo 5 demonstra uma análise mais detalhada de um circuito testado. O capítulo 6 discute os resultados alcançados na verificação de circuitos digitais. O Capítulo 7 mostra as conclusões e trabalhos futuros. Nos Anexos A e B são demostrados alguns gráficos de simulações temporais e da taxa de cobertura dos circuitos testados. Já no Anexo C são descritos os circuitos testados. Finalmente no Anexo D é mostrado uma publicação desse trabalho (HENNES e SANTOS R.R.; MOLZ 2009) e no E é mostrado o projeto da placa de circuito impresso desenvolvida usando um ARM e um barramento para a placa do FPGA projetada.

2 CONCEITOS BÁSICOS

Modelos têm sido a base de toda a ciência humana ao longo dos séculos. Antes do estabelecimento dos modelos que representam o átomo, as reações químicas e as propriedades da matéria, a química não existia. Como seria o mundo se um dia não houvessem sido definidos modelos matemáticos para resolver problemas abstratos, tais como a álgebra, geometria, trigonometria, entre outros. Destes modelos matemáticos se desenvolveram inúmeras ciências, como a física e a engenharia.

Na área de testes de circuitos integrados, um modelo muito importante é o modelo de falhas. Serão mostrados diversos modelos de falhas nesse capítulo, porém somente o modelo de falhas *stuck-at* descrito na seção 2.3.1 é adotado durante todo o trabalho. Além dos tipos de modelos são vistas algumas definições usadas na área de testes, algumas formas de se testar um circuito integrado e sua exata compreensão se faz necessária para o entendimento dos próximos capítulos.

2.1 Defeito, Erro e Falha

Os conceitos de defeito (*defect*), erro (*error*) e falha (*fault*) são usados de forma confusa na literatura de testes de circuitos. Nesse trabalho, essas definições serão usadas conforme as definições descritas a seguir (BUSHNELL M. L.; AGRAWAL 2000).

- Defeito - Um defeito em um sistema é a diferença entre o hardware desenvolvido e o que se pretendia desenvolver. Alguns defeitos comuns encontrados são: falta da janela de contato, transistores parasitas, degradação de contatos, etc. É importante observar que o defeito pode surgir tanto durante a fabricação, quanto durante o uso

- Falha - A abstração de um defeito em nível de funções (modelos físicos ou lógicos do dispositivo)
- Erro - Um erro é causado quando existe um sinal errôneo na saída de um circuito causado por um circuito defeituoso (com falha)

Esses conceitos podem ser melhor compreendidos com um exemplo de um circuito digital que consiste de duas entradas A e B, uma saída Z e uma porta lógica[1] AND. Um defeito possível pode ser a falta de um contato em umas das entradas da porta AND, conforme mostrado na Figura 3. A representação deste defeito em nível lógico é ter sempre o valor lógico zero (*stuck-at-0*), independente do valor colocado neste sinal. Dessa forma, se a entrada do circuito recebesse A=1 e B=1, a falha iria mostrar um erro na saída do circuito. A saída Z iria mostrar o nível lógico 0, enquanto o correto seria 1. Deve-se notar que o erro desse circuito não é sempre visível. Se umas das entradas receber nível lógico 0 o erro não apareceria.

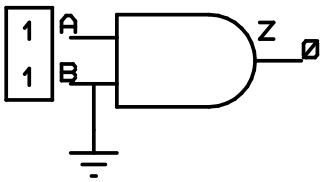


Figura 3: Porta lógica AND

- Defeito - *stuck-at-0*
- Falha - sinal B permanentemente com valor lógico 0
- Erro - Z apresenta um valor errado se A = B = 1

2.2 Teste estrutural e Teste funcional

O teste funcional de um circuito visa verificar se todas as funções desse circuito digital estão sendo executadas corretamente. No caso de processadores, o teste funcional necessita cobrir as diferentes funções implementadas no conjunto de instruções do processador, por exemplo. Para isso é necessário somente o conjunto de instruções da arquitetura[2] (ISA - *Instruction Set*

[1] São dispositivos que operam um ou mais sinais lógicos de entrada para produzir uma e somente uma saída, dependente da função implementada no circuito.

[2] É uma lista de todas as instruções e suas variações que um processador pode executar.

Architecture) para se desenvolver os estímulos de teste, não necessitando de nenhum outro modelo de mais baixo nível como um descrição em portas lógicas (MORAES 2006). A testabilidade de um teste funcional depende do conjunto de instruções usados para testar as funções do processador. Na maioria dos casos, operandos pseudo-aleatórios são usados no teste funcional do circuito, levando a testes com tempos excessivos e não conseguindo uma boa cobertura de falhas estruturais. As seqüências de teste desenvolvidas para a verificação do circuito, podem ser reusadas, reduzindo o custo do desenvolvimento do teste (GIZOPOULOS D.; PASCHALIS 2004). Já o teste estrutural depende de uma estrutura específica (portas lógicas, *netlist*) do circuito. Umas das grandes vantagens do teste estrutural é que ele nos permite desenvolver algoritmos. Esses algoritmos são baseados no modelo de falhas (BUSHNELL M. L.; AGRAWAL 2000).

Tabela 1: Vantagens e desvantagens entre testes

	Vantagens	Desvantagens
Teste Funcional	-Sem necessidade de detalhes de baixo nível -Reuso de padrões de teste de verificação funcional -Baixo custo de desenvolvimento de teste	-Sem relação com falhas estruturais -Baixa cobertura de defeitos -Longos programas de teste -Operando pseudo-aleatórios -Longas sequências de teste
Teste Estrutural	-Rápidos programas de teste -Uso de ferramentas EDA -Alta cobertura de falhas -Pequenas seqüências de teste	-Necessidade do modelo lógico do processador -Mais alto custo do desenvolvimento do teste

Fonte: (GIZOPOULOS D.; PASCHALIS 2004, p. 59)

Ferramentas de EDA[3] (*Electronic Design Automation*) podem ser usadas para a geração automática dos vetores de testes[4], com possível suporte à técnicas de DFT (*Design for Testability*) como a controlabilidade e observabilidade. A geração do teste estrutural só pode acontecer se o modelo em níveis de portas lógicas for disponibilizado (GIZOPOULOS D.; PASCHALIS 2004). No caso de se ter essa informação, uma alta taxa de cobertura de falhas é possível com uma pequena quantidade de vetores. A Tabela 1 resume as vantagens e desvantagens dos dois tipos de testes. Com base nas informações contidas na Tabela 1, uma das grandes vantagens dos testes estruturais está na alta cobertura de testes e na rapidez desses programas de teste. Dessa forma, este trabalho esta focado em testes estruturais.

[3]É uma categoria de ferramentas de CAD para desenvolvimento e produção de sistemas eletrônicos variando de placas de circuito impresso até circuitos integrados.

[4]Ná area de testes de circuitos integrados a geração automática de vetores é conhecida como ATPG (*Automatic Test Pattern Generation).*

2.3 Modelo de Falhas

A modelagem de falhas está intimamente ligada à modelagem do circuito. Na hierarquia de projeto, níveis mostram os graus de abstração do circuito. Um caso que demonstra poucos detalhes é o nível comportamental (alto-nível). Portanto modelos de falhas usando esse nível comportamental não podem ter uma correlação direta com defeitos de fabricação do circuito. O nível RTL (*Register-Transfer Level*) ou nível lógico é composto por uma descrição do circuito em portas lógicas através de um *netlist*. O modelo de falhas *stuck-at* é o modelo mais usado em testes de circuitos digitais. Ainda no nível lógico existem os modelos de falha de atrasos e de *bridging* (BUSHNELL M. L.; AGRAWAL 2000). Já em nível de transistor é possível ver os tipos de falhas *stuck-open*, muito conhecidas como falhas dependentes da tecnologia. Além dos modelos citados, ainda existem os modelos de falhas que não se enquadram em nenhuma das hierarquias de projeto. Um exemplo típico são as falhas de corrente quiescente (I_{DDQ}), relevantes para a tecnologia CMOS[5] (*Complementary Metal Oxide Semiconductor*) (MORAES 2006).

2.3.1 Falhas *Stuck-at*

Falhas lógicas são modeladas como falhas *stuck-at*. Uma falha *stuck-at* força um valor fixo (0 ou 1) para uma linha de sinal de um circuito, onde essa linha pode ser a entrada ou saída de uma porta lógica ou *flip-flop*[6] (BERGER I.; KOHAVI 1973). Esse *stuck-at* pode ser um *stuck-at-1* (s-a-1) ou um *stuck-at-0* (s-a-0). O modelo *stuck-at-0* é simples e independe da tecnologia usada. Uma linha de sinal de um circuito pode ter um s-a-0, s-a-1 ou livre de falhas (*fault-free*). Um circuito com n linhas de sinal pode ter 3^{n-1} possibilidades de linhas *stuck-at*. Então para um circuito de tamanho moderado já é possível ter um grande número de falhas *stuck-at*. Por isso é uma prática comum considerar somente o modelo de falhas *single stuck-at* . Esse modelo assume que somente uma entrada ou saída de uma porta lógica ou *flip-flop* pode estar s-a-0 ou s-a-1. O número máximo de falhas *single stuck-at* para um circuito com n linhas é de $2 * n$ (BUSHNELL M. L.; AGRAWAL 2000).

O número de falhas usando o modelo de falhas *single stuck-at* é consideravelmente reduzido. Considerando que se tem uma porta lógica NAND com uma entrada A com s-a-1,

[5] É um tipo de circuito integrado onde se incluem elementos de lógica digital (portas lógicas, *flip-flops*, contadores, decodificadores, etc).

[6] É um circuito digital pulsado capaz de servir como uma memória de um *bit*.

conforme mostrado na Figura 4, essa entrada sempre permanecerá em nível lógico 1, independentemente da lógica colocada nas suas entradas. A saída dessa NAND é 0 quando a entrada B estiver em 1 e quando o s-a-1 em A estiver presente. Já se esse circuito não possuir o s-a-1 ou s-a-0, ou seja, for *fault-free*, a saída desse circuito será 1 para as entradas mostradas na Figura 4. Então, se for usado na entrada desse circuito A=0 e B=1, pode-se testar o s-a-1 em A, porque existe uma diferença nas saídas do circuito *fault-free* e o *faulty gate* (LALA 1997).

Devido ao que foi descrito acima (modelo simples e independente da tecnologia), foi escolhido para a verificação o modelo de falhas *single stuck-at*. Dessa forma toda a metodologia usada neste trabalho esta baseada no modelo de falhas *single stuck-at*.

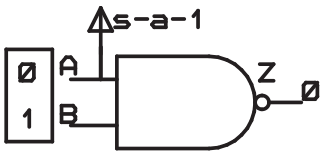


Figura 4: Porta lógica NAND

2.3.2 Equivalência de falhas *single stuck-at*

Para um circuito existem $2 * n$ falhas *single stuck-at*, porém muitas dessas falhas são equivalentes. Se for analisado um circuito como se fossem simples portas lógicas, então é possível analisar grandes circuitos. Considerando uma porta lógica AND conforme visto na Figura 5(a), qualquer falha do tipo s-a-0 na saída ou entrada dessa porta sempre terá uma saída constante em 0. Dessa forma é possível dizer que essas falhas s-a-0 são equivalentes. Uma análise similar para outras portas lógicas é possível ver na Figura 5 (BUSHNELL M. L.; AGRAWAL 2000). No caso de se ter uma porta lógica OR conforme Figura 5(d), qualquer falha do tipo s-a-1 na saída ou entrada dessa porta, sempre terá uma saída constante em 1. Já tendo uma NAND conforme Figura 5(b), qualquer falha s-a-0 em uma entrada será equivalente a um s-a-1 na saída e um s-a-0 na outra entrada. Na Figura 5(c) é visto uma NOT, no caso uma s-a-1 na entrada é equivalente a um s-a-0 na saída e vice-versa. No caso de se ter uma NOR conforme Figura 5(e), um s-a-1 em uma das entradas será equivalente a um s-a-0 na saída e um s-a-1 na outra entrada.

Na Figura 6 é mostrado um circuito com falhas equivalentes. A redução da lista de falhas (*fault collapsing*) é feita analisando-se as equivalências entre entradas e saídas e retirando as

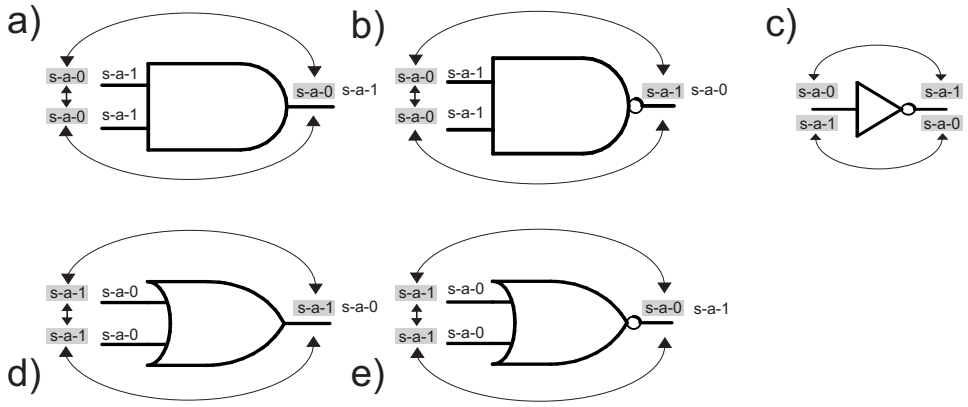


Figura 5: Equivalência de falhas

falhas equivalentes conforme Figura 6. Na Figura 5(a) foi mostrado que uma AND possui 3 falhas s-a-0 equivalentes, ou seja, só é necessário deixar a análise de uma delas no circuito. No caso das portas lógicas AND mostradas na Figura 6, foram excluídas duas falhas equivalentes nas entradas. Já no caso da porta lógica OR mostrada na Figura 6, foram excluídos duas falhas s-a-1 de acordo com a equivalência vista na Figura 5(d). O processo de selecão de uma falha a partir de cada conjunto de falhas que apresentam o comportamento equivalente é chamado de redução da lista de falhas (*fault collapsing*). O conjunto dessas falhas é conhecido como *equivalence collapsed set*. O tamanho relativo da *equivalence collapsed set* com o total de falhas é conhecido como *collapse ratio* e a equação 2.1 mostra a sua relação.

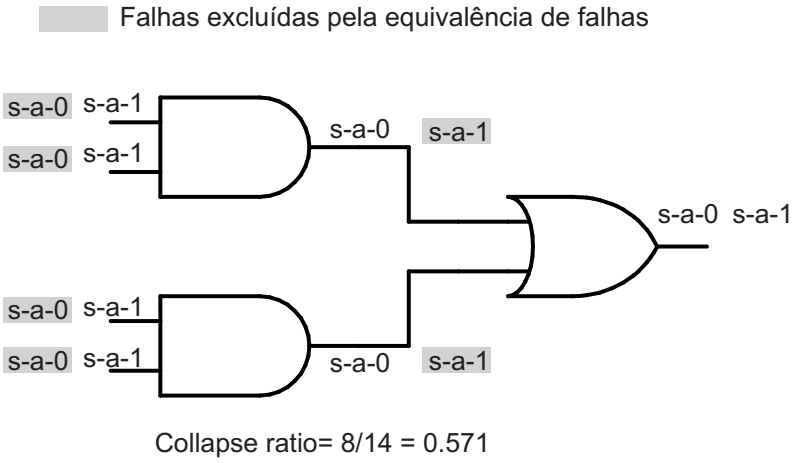


Figura 6: Exemplo de equivalência de falhas

$$Collapse\,ratio = \frac{Set\,of\,collapsed\,faults}{Set\,of\,all\,faults} \tag{2.1}$$

Além da equivalência de falhas ainda existe a dominância de falhas (*fault dominance*) con-

forme mostrado na Figura 7. Uma falha f_1 e uma falha f_2 é chamada de equivalente, se qualquer teste que detecta f_1 também detecta f_2 e vice-versa. É dito que uma falha f_1 domina uma f_2, se qualquer teste que detecta f_2 também detecta f_1, porém, o inverso não é verdadeiro (RAIK 2001). Um exemplo de equivalência de falhas e dominância de falhas usando o modelo *stuck-at* é discutido na seção 3.3.1.

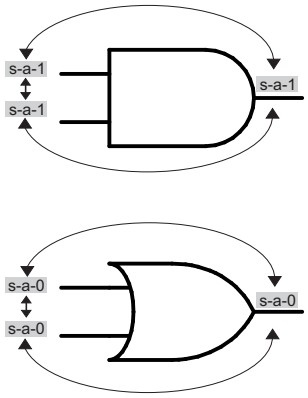


Figura 7: Exemplo de dominância de falhas

2.3.3 Falhas de *Bridging*

Este modelo de falhas demonstra o comportamento de falhas em que uma linha do circuito influência o valor lógico de outra linha do circuito, devido a uma ligação resistiva entre os dois. Essas falhas conseguem ser modeladas tanto a nível de transistor, como a nível lógico. As falhas de *bridging* representam um curto-circuito entre um determinado grupo de sinais. Este tipo de falhas ocorre entre nós fisicamente muito próximos entre si, no *layout* do circuito. Quando uma falha de *bridging* é ativada há uma linha agressora que vai forçar o valor lógico de outra linha, chamado de linha vítima. O valor lógico desse curto-circuito pode ser modelado como 0 dominante (*wired*-AND), como um 1 dominante (*wired*-OR), ou indeterminado, dependendo da tecnologia usada (LALA 1997).

2.3.4 Falhas de Atraso

O objetivo do teste de atraso é de detectar defeitos no atendimento das especificações temporais. Esse tipo de falha é visto quando os parâmetros do projeto não atendem a relação temporal entre entradas e saídas do circuito. A necessidade de testes de atraso é um problema comum enfrentado pela indústria de semicondutores. A necessidade desses testes é vista quando

um circuito funciona adequadamente com uma freqüência baixa de *clock*[7] e falha com uma alta frequência de *clock* (KRSTIC A.; CHENG 1998). Um exemplo de falha de atraso é mostrado na Figura 8, onde existe um pulso na entrada da porta lógica NOR e o sinal deve levar um certo tempo até a saída da porta. Um sinal nessa porta sem atraso é mostrado na Figura 8(a) e o sinal com atraso é mostrado na Figura 8(b).

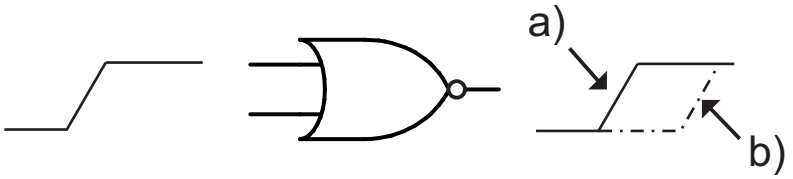


Figura 8: Exemplo de uma falha de atraso

A crescente necessidade por testes de atraso é o resultado no avanço na fabricação de circuitos integrados. Testes de atraso são defeitos muito complexos. Existem dificuldades para a geração de testes e a aplicação desses testes. Alguns dos principais tipos de problemas no teste de atraso são descritos abaixo (KRSTIC A.; CHENG 1998):

- Os defeitos de atraso podem ser ativados e observados pela propagação do sinal através do circuito
- O teste de atraso em circuitos de alta velocidade requer o uso de testadores de alta velocidade. Porém, devido a seu alto custo, a maioria dos testadores disponíveis são mais lentos que os novos circuitos que necessitam ser testados
- Pequenos defeitos de atraso podem ser modelados usando o modelo do caminho de falhas de atraso. Porém, os novos circuitos possuem uma grande quantidade de caminhos e apenas uma pequena fração destes pode ser testada
- O processo de fabricação de circuitos integrados introduziu novas dificuldades devido a seus circuitos serem mais suscetíveis a ruídos[8]
- O *design* e a síntese usando técnicas para melhorar a testabilidade de falhas de atraso resultam em um grande aumento da área e queda em performance, e um aumento da quantidade de entradas do circuito

[7] É um sinal usado para coordenar as ações de dois ou mais circuitos eletrônicos.

[8] É todo fenômeno aleatório que perturba a transmissão correta das mensagens e que geralmente procura-se eliminar ao máximo.

Dentro dos testes de atraso existe um tipo de teste chamado de teste *at-speed*. Os vetores de teste no teste *at-speed* são aplicados e observados na mesma freqüência de operação normal do circuito. Os testes *at-speed* são testes já necessários nos modernos projetos eletrônicos. Grandes velocidades de *clock* e circuitos com áreas menores são encontrados em circuitos eletrônicos atuais, devido a isso, existe um aumento de defeitos relacionados a velocidade. O testes *at-speed* mais comuns para checar defeitos de fabricação e variação de processo incluem vetores de teste criados para modelos de falha de transição e de atraso de caminho (SWANSON B. ; GOSWAMI 2006).

2.4 Testes de I_{DDQ} (*Quiescent Current*)

O modelo de falhas I_{DDQ} permite detectar defeitos de fabricação através da análise de corrente[9] em circuitos CMOS. Os testes para este tipo de falhas são baseados na corrente durante um intervalo de tempo. Na Figura 9(a) é mostrado um circuito inversor que está sendo testado usando I_{DDQ}. Desta forma é monitorada a corrente quiescente (I_{DDQ}), conforme apresentado na Figura 9(c), do circuito que está sendo testado, enquanto se excitam as entradas desse circuito. A excitação das entradas do circuito é mostrado na Figura 9(b). No caso de ser verificado que a corrente quiescente se mantém elevada, durante uma situação estável, significa que este circuito é defeituoso (SABADE S.S. ; WALKER 2004). Como este método é simplificado, o processo de geração de teste, não necessita mais propagar as falhas para uma saída primária (LUBASZEWSKI M.; COTA 2002).

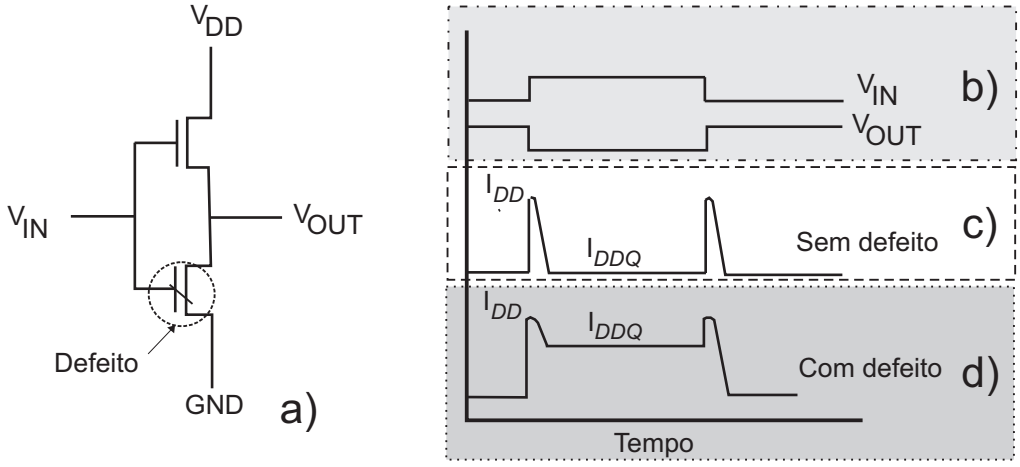


Figura 9: O princípio do teste de I_{DDQ}

[9] É o fluxo ordenado de partículas portadoras de carga elétrica.

No caso de ausência de defeitos, e quando a entrada é estável, a corrente de repouso que passa do VDD para o GND é baixa, como mostrada na Figura 9(c). Na presença de um defeito, conforme apresentado na Figura 9(d), uma corrente mais alta flui pelo transistor. Dessa forma é possível identificar um circuito defeituoso medindo a corrente de fuga (SABADE S.S. ; WALKER 2004).

2.5 *Automatic Test Equipment (ATE)*

O ATE é um aparelho usado para aplicar vetores de testes a um DUT (*device-under-test*) e analisar as respostas do DUT conforme mostrado na Figura 10, permitindo marcar esse DUT como defeituoso ou não defeituoso (BUSHNELL M. L.; AGRAWAL 2000). Obviamente a qualidade do teste vai depender de uma escolha de bom vetores, ou seja, que consigam uma maior taxa de cobertura. Para cada DUT produzido são repetidos as etapas de aplicação, captura e análise das respostas de testes. Por isso, é importante que o tempo gasto seja o menor possível.

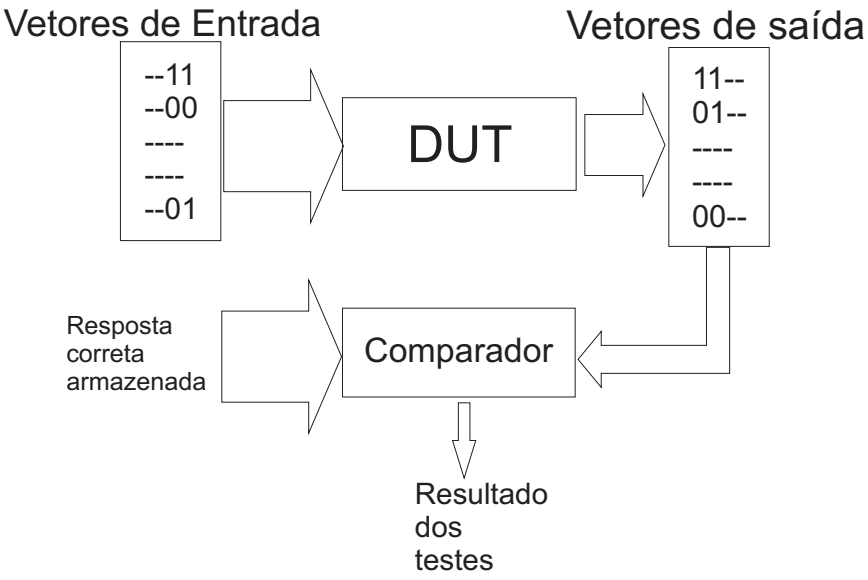


Figura 10: Princípio de testes

Os modernos ATE possuem modernos processadores controlados por um programa descrito em linguagem de alto nível. De uma maneira geral, cada circuito integrado pode ser testado de duas maneiras (BUSHNELL M. L.; AGRAWAL 2000):

- Testes paramétricos: testes paramétricos DC incluem teste de curto-circuito, teste aberto, corrente máxima de teste, etc.Testes paramétricos AC incluem teste de atraso, teste de

tempo de subida e descida, etc

- Testes funcionais: consistem em colocar vetores na entrada e verificar as respostas. Esses testes cobrem uma alta porcentagem de falhas modeladas em circuitos. Os vetores funcionais usados nesses testes, podem ser entendidos como vetores de verificação, que são usados para verificar se o hardware esta de acordo com a especificação. Porém, no mundo dos ATE, qualquer vetor aplicado pode ser entendido como vetor de falhas funcionais aplicados durante a fabricação. Esses dois tipos de testes funcionais podem ser os mesmos ou não

Para testes *off-chip* o ATE (*automatic test equipment*) é necessário para aplicar o teste. A parte do *hardware* do ATE, inclui um sistema de computação com suficiente espaço para os vetores de teste e as respostas esperadas. De acordo com a função que esta sendo testada, o testador é classificado como testador digital, memória ou analógico. Testadores digitais de circuitos integrados de lógica e de memória são tratados de forma diferente. Testadores de memórias geralmente são testados em paralelo, com a mesma entrada para diversas partes e suas saídas lidas juntamente. Enquanto que testadores de lógica necessitam que os vetores de testes sejam testados um por vez. Testadores digitais consistem de testadores de memória e de testadores de lógica. A grande diferença entre testadores digitais e analógicos é mostrado na configuração individual dos pinos. Pinos digitais são facilmente suportados por qualquer outro circuito digital. Já os pinos analógicos possuem uma grande variação de funções.

Combinando circuitos digitais e analógicos no mesmo circuito integrado, é necessário um grande cuidado para coordenar e isolar os dois tipos de sinais e terras. Por isso, os testadores de circuitos mistos possuem um custo muito mais elevado por pino do que outros testadores. Para amortizar esse custo é realizado o teste em uma grande quantidade de produção. Além disso, é mantido um custo menor deixando o tempo o menor possível. Para isso é necessário manter a quantidade de vetores de teste o menor possível (MOURAD S. ; ZORIAN 2000).

2.6 Auto Teste (BIST)

Auto-teste significa a capacidade de um circuito integrado de testar-se a si próprio. O auto-teste de um processador, significa que os vetores de teste são aplicados nele e as respostas

são compararadas para ver se houve um correto funcionamento sem o uso de um ATE externo, porém usando recursos internos do processador (GIZOPOULOS D.; PASCHALIS 2004). Um comparativo entre auto-teste e testadores externos é mostrado na Tabela 2.

Tabela 2: Testador externo x Auto-teste

	Vantagens	Desvantagens
Testador Externo	-Pequeno aumento de hardware no *chip* -Pequeno impacto na performace do *chip*	-Teste não é feito na velocidade de funcionamento -Alto custo ATE
Auto-teste	-Teste na velocidade de funcionamento -Baixo custo do ATE -Reutilizável durante o ciclo de vida do produto	-Aumento de hardware -Queda de performace

Fonte: (GIZOPOULOS D.; PASCHALIS 2004, p. 57)

Com o aumento da miniaturização dos circuito integrados, o uso de ATE acaba se tornando problemático. As desvantagens do ATE em relação ao BIST são descritas abaixo (JERVAN 2005):

- O ATE é um equipamento extremamente caro
- Com o aumento da complexividade dos circuitos integrados, o tempo de teste continua aumentando e o *time to market* esta se tornando inaceitável
- O tamanho do ATE e conseqüentemente os requisistos de memória aumentam constantemente
- A freqüência de operação do ATE deve ser maior ou igual a freqüência do DUT

2.7 *Design For Test* (DFT)

O chamado *design for test*, ou projeto visando a testabilidade são algumas técnicas usadas para se melhorar a testabilidade de um circuito. Mesmo possuindo uma boa ferramenta para a geração de vetores de testes, falhas difíceis de detectar impedem que uma boa relação entre cobertura de falhas e o tempo de aplicação desses testes seja alcançada. Nesses casos uma solução possível seria re-projetar o circuito visando a testabilidade, ou seja, melhorar a acessibilidade de partes do circuito difíceis de testar (LUBASZEWSKI M.; COTA 2002).

De uma forma geral, esses métodos de DFT, melhoram a capacidade de se observar as saídas e o comportamento dos nodos internos (observabilidade) melhorando também a capacidade de se levar sinais da entrada do circuito até nodos internos do circuito (controlabilidade). A controlabilidade de uma linha se consegue impondo nessa linha um valor lógico 0 ou 1. Num circuito integrado o acesso físico é feito pelos pinos de entrada e saída, portanto torna-se claro que baixos índices de controlabilidade tornarão difícil a tarefa de impor um valor lógico à nossa escolha, numa linha interna qualquer. No entanto é uma tarefa fundamental no procedimento de geração de testes poder detectar uma falha nessa linha. Um vetor só nos permitiria a detecção da falha se o sinal de erro nessa linha pudesse ser propagado até uma saida primária. Portanto, a menor ou maior facilidade de se propagar esse valor até a saída do circuito se dá o nome de observabilidade (FERREIRA 1998).

2.8 Conclusões

Nesse capítulo foram vistos alguns conceitos básicos na área de testes e alguns métodos e modelos de testes. O método de verificação usado neste trabalho foi o teste estrutural e de acordo com a Tabela 1, existem vantagens como o uso de EDA e a alta cobertura de falhas. Já o modelo *single stuck-at* foi utilizado devido a sua simplicidade, grande utilização e independência da tecnologia.

Neste trabalho foram usadas duas ferramentas de EDA, uma chamada de Leonardo Spectrum (Mentor Graphics) e a outra de Turbo Tester (JERVAN G. ; MARKUS). Conjuntamente a essas ferramentas EDA foi usada a linguagem de programação JAVA.

Na metodologia da plataforma de verificação desenvolvida usando as EDAs não se teve a pretensão de se igualar com um moderno ATE e sim desenvolver uma plataforma de verificação de baixo custo que seja mais acessível para empresas de menor porte. Além disso a plataforma desenvolvida realiza verificação em software e hardware baseado no modelo de falhas *stuck-at* e não efetuando outros tipos de testes como por exemplo testes paramétricos.

3 GERAÇÃO DE TESTES PARA CIRCUITOS DIGITAIS

Para se analisar um circuito digital é necessário um modelo matemático para representá-lo. O modelo matemático usado nesse trabalho é o SSBDD (*Structurally Synthesized Binary Decision Diagrams*) devido a ferramenta Turbo Tester usá-lo e esse ser um modelo bem aceito, diferentemente de outros modelos BDD (*Binary Decision Diagrams*). Com o modelo matemático escolhido para representar o circuito, ainda é necessário o modelo do tipo de falhas a ser testado. Neste caso escolheu-se o modelo de falhas *single stuck-at*. Com o modelo matemático escolhido e o modelo da simulação de falhas, a geração automática de vetores é o caminho ideal para se conseguir um procedimento de teste eficiente para um determinado circuito. A geração automática de testes será vista nesse capítulo e tenta conseguir a máxima cobertura de falhas usando vetores de testes nas entradas e analisando as saídas do circuito. A geração desses vetores de testes, no domínio digital, foi feita através de algortimos determinísticos. Os algoritmos D (ROTH 1966), PODEM (GOEL 1981), GENETIC (WHITLEY 1994), RANDOM (MOURAD S. ; ZORIAN 2000) serão estudados nesse capítulo. O algoritmo D proposto por Roth, precursor na área de testes, é o fundamento do algoritmo PODEM usado nesse trabalho. Além desse algoritmo, também foram usados os algoritmos GENETIC e RANDOM.

3.1 Simulação de Falhas

A simulação de falhas (*fault simulation*) consiste no modelamento do circuito na presença de falhas. Comparando a resposta do circuito com falhas e do circuito sem falhas, usando o mesmo vetor de teste, pode-se determinar as falhas detectadas por esse vetor (CILETTI 2002). A simulação de falhas classifica as falhas em um circuito como **detectáveis** ou **indetectáveis**. A lista de todas as falhas é feita com o modelo *single stuck-at* usando a equivalência de falhas discutida na seção 2.3.2. A simulação é inicializada com os vetores de verificação fornecidos. A

simulação produz uma lista de falhas. A cobertura dessas falhas, ou seja, a cobertura de falhas é mensurada através do número de falhas detectadas e do total de falhas que consta na lista de falhas conforme mostrado na equação 3.1.

$$Cobertura\,de\,falhas = \frac{Quantidade\,de\,falhas\,detectadas}{Total\,de\,falhas} \tag{3.1}$$

Quando se alcança a cobertura de falhas desejada, a simulação pode terminar. Se existirem vetores que não podem ser simulados, estes podem ser removidos para diminuir a lista de vetores de teste. Porém, se esses vetores de teste não produzirem uma cobertura adequada conforme a equação 3.1, algumas dessas falhas não detectadas da lista de falhas podem ser passadas a um programa de testes para se produzir novos vetores (BUSHNELL M. L.; AGRAWAL 2000). Na Figura 11 é mostrado um circuito C() que é livre de falhas e um circuito C(f_1) até um C(f_n) são cópias com falhas inseridas. O mesmo vetor é aplicado para todos os circuitos e as saídas dos circuitos com falhas são comparadas com o circuito livre de falhas.

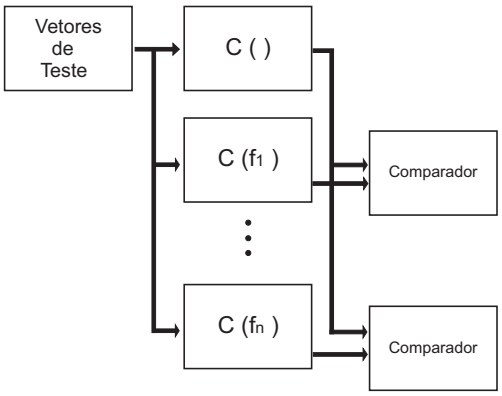


Figura 11: Simulação de falhas

Existem diversos métodos de simulação de falhas como o serial, paralelo, dedutivo e concorrente. A simulação de falhas serial é o método mais lento e mais simples de todos, porém usa menos memória. É baseada na simulação de um circuito livre de falhas (*fault free*) e um circuito com a presença de uma falha e é comparado às respostas do circuito livre de falhas e o com falha. No caso da resposta ser diferente, a falha é detectada. O processo é repetido em seqüência para todas as falhas. O tempo de execução é proporcional ao número de falhas no circuito. A simulação de falhas paralela simula simultaneamente o circuito sem falhas e um número fixado de circuitos com falhas simultâneas é mais rápido que a simulação serial, porém necessita de mais memória e um código mais complexo (AL-ASAAD 1998). Já a simulação con-

corrente é o algoritmo mais usado e limita a procura da simulação de falhas na parte relevante do circuito (CILETTI 2002).

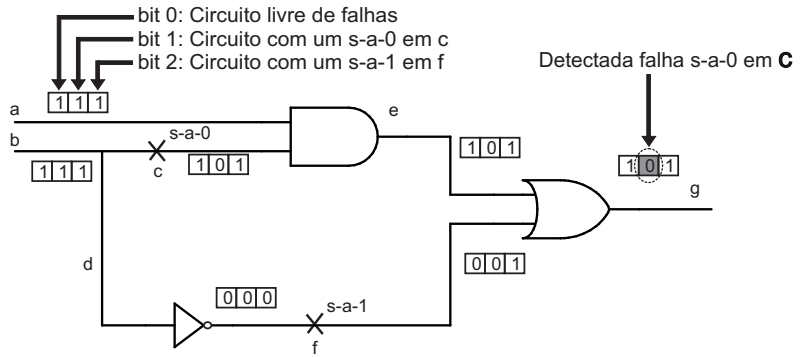


Figura 12: Simulação de falhas paralelas
Adaptado de (BUSHNELL M. L.; AGRAWAL 2000, p. 108)

Na Figura 12 é mostrado um circuito que está sendo simulado, usando-se simulação paralela para em **c** um *stuck-at-0* e **f** um *stuck-at-1*. O primeiro bit representa o circuito sem falhas e os outros dois representam os circuitos com falhas. O primeiro bit em todas as caixas de simulação é o do circuito livre de falhas, o segundo bit das caixas de simulação é o do **c** com um s-a-0 e o terceiro bit é o do **f** com o s-a-1. No circuito sem falhas, ou seja, no primeiro bit, são inseridos os valores a = 1 e b = 1 nas entradas. Com isso é esperado em **e** um bit 1 e em **f** um bit 0, conforme mostrado na Figura 12. Em **g** é esperado o valor de 1 para o circuito livre de falhas. Agora usando o mesmo circuito com uma falha s-a-0 ativada em **c** e é visto no bit 1. Com a entrada mantida como anteriormente, ou seja, a = 1 e b = 1; em um circuito livre de falhas a saída em **e** seria 1, porém como **c** possui um s-a-0 a saída em **e** fica com 0. Com o valor de **f** em 0 e de **e** em 0 a saída do circuito em **g** será 0. Comparando o circuito livre de falhas e o com falha em c, é possível detectar a falha. Já no caso de uma falha em f, não será detectável essa falha pelo bit 3, conforme apresenta-se na Figura 12.

3.2 Definição de *Automatic Test-Pattern Generation* (ATPG)

Para se testar um circuito digital, aplicam-se estímulos nas suas entradas e são analisadas as suas saídas. A geração desses estímulos, mais o cálculo de suas saídas são chamados de *test pattern generation*. Esses *test patterns* são gerados por uma ferramenta denominada *automatic test pattern generation* (ATPG) e podem ser aplicados a um circuito usando *automatic test equipment* (ATE) (JERVAN 2005).

Algoritmos ATPG simulam uma falha no circuito, e são usados métodos para que essa falha se propague pelo circuito e acabe sendo mostrada na saída. A saída desse sinal é alterada se o circuito possui uma falha. As falhas são propagadas em portas lógicas AND e NAND, colocando as outras entradas em nível lógico 1. Já em portas lógicas OR e NOR as falhas são propagadas colocando as outras entradas em nível lógico 0. Em portas lógicas XOR e XNOR colocando as outras entradas em 0 ou 1 conforme o caso (BUSHNELL M. L.; AGRAWAL 2000).

3.3 Modelo BDD (*Binary Decision Diagrams*)

Com o constante aumento da integração de circuitos integrados em circuitos digitais modernos se faz necessário métodos e algoritmos eficientes. A eficiência de qualquer algoritmo depende do modelo matemático usado. Durante muitas décadas foram procurados modelos matemáticos para representação de funções booleanas. Um dos primeiros modelos aceitos foi o BDD várias décadas atrás. Na Figura 13(a) é mostrado um circuito digital e na Figura 13(b) é mostrado esse circuito usando o modelo BDD.

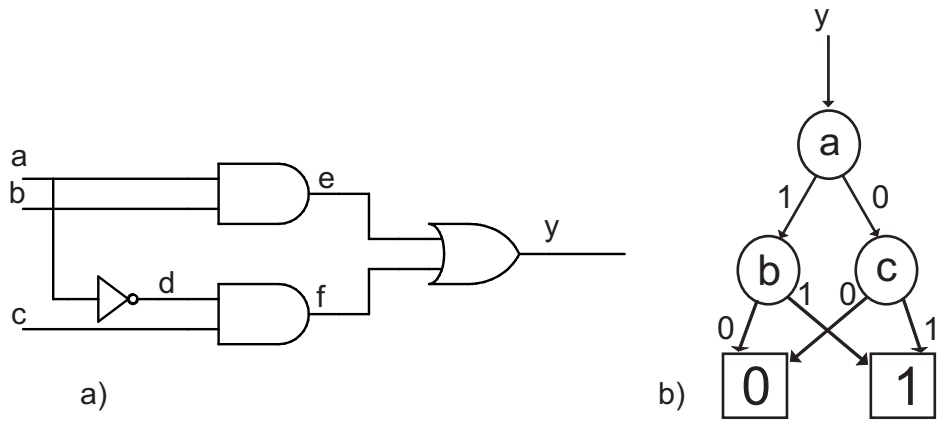


Figura 13: Representação de um circuito lógico usando o modelo BDD

Bryant (BRYANT 1986) propôs uma nova estrutura chamada *Reduced Ordered Binary Decision Diagrams* (ROBDDs). Porém esse método é limitado em grandes circuitos devido ao alto uso de memória. Durante a última década, muitas modificações do modelo BDD foram propostas. Como resultado, modelos similares apareceram para representar qualquer circuito booleano. Um modelo que surgiu na área de testes de circuitos digitais foi o SSBDD (*Structurally Synthesized Binary Decision Diagrams*). Esse modelo compacto preserva a informação estrutural do circuito modelado. Além desse modelo existem outros modelos matemáticos, porém o SSBDD possuí vantagens no domínio da simulação lógica. O tempo para se gerar um modelo

SSBDD é linear, respeitando o número de portas lógicas, já outro modelo, o ROBDD, tem um comportamento exponencial. Um segundo detalhe a se considerar, o modelo SSBDD possui um tamanho linear em relação ao tamanho do circuito, enquanto o modelo ROBDD possui um tamanho exponencial. Outro fator muito importante, é que o modelo SSBDD preserva as informações estruturais do circuito (AND, OR, NOT) enquanto que outros modelos não o fazem (JUTMAN A. ; RAIK).

O modelo SSBDD é a representação da lógica do circuito digital onde as funções booleanas que definem um circuito combinacional como uma rede de funções booleanas básicas, como, AND, NAND, OR, NOR, INVERTER e BUFFER. Essa técnica funciona para qualquer circuito combinacional que seja especificado por essas 6 básicas portas lógicas. Circuitos que contenham outros tipos de portas, devem ser convertidos a esses 6 tipos antes que se gere o modelo SSBDD (JUTMAN).

Devido ao que foi descrito, foi utilizado o modelo SSBDD para a geração dos testes de circuitos. Para se chegar até esse modelo é necessário ter o arquivo descrito em um *netlist* do circuito. Com esse *netlist*, é possível se converter para o modelo SSBDD, que será usado para se testar o circuito. A geração desses arquivos será melhor explicado no capítulo 4.

3.3.1 Falhas *Stuck-At* no modelo SSBDD (*Structurally Synthesized Binary Decision Diagrams*)

Considerando o circuito da Figura 14, que possui 9 linhas e 18 falhas *stuck-at* (2 falhas por cada linha). Porém, se for aplicadas a equivalência e a dominância nessas falhas, conforme descrito na seção 2.3.2, sobrariam somente 8 falhas para se testar. Na Figura 14 são mostradas todas as falhas s-a-0 e s-a-1 que necessariam ser testadas (RAIK 2001).

Na porta AND (1) são retiradas por equivalência as falhas s-a-0 em **c** e s-a-0 em **d**, conforme Figura 5(a) da página 24 que foi aqui inserida novamente com o nome de Figura 15(a), para facilitar a explicação desse ponto. Usando a mesma lógica, para a porta AND (2) foi retirado em **e** o s-a-0 e em **h** o s-a-0. Para a porta lógica OR foram retirados por equivalência as falhas s-a-1 em **f**, s-a-1 em **b** e s-a-1 em **g**, conforme Figura 15(d). Já na porta NOT foram retiradas as falhas s-a-0 em **a** e o s-a-1 em **f** conforme Figura 15(c).

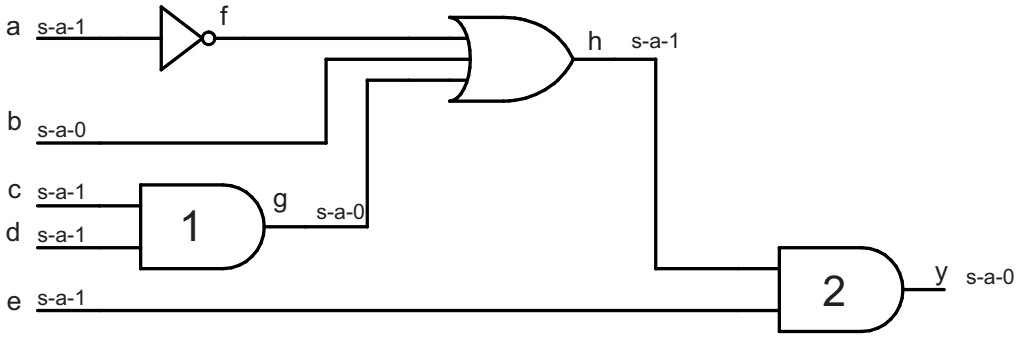


Figura 14: Circuito digital com suas *collapsed faults*

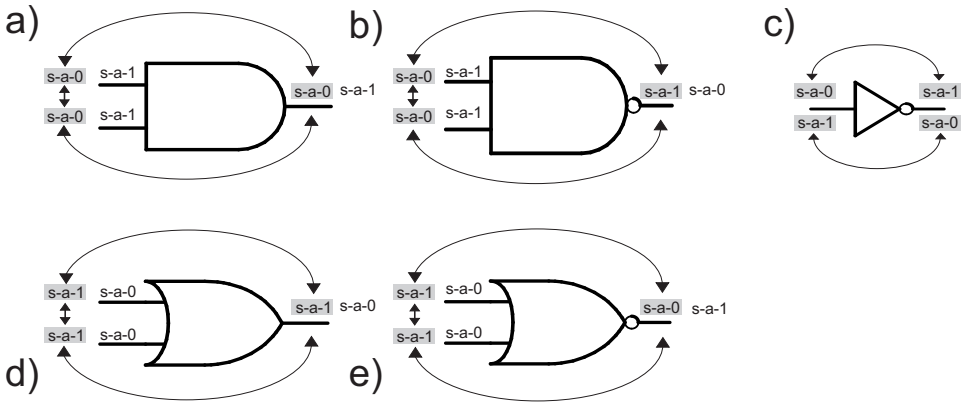


Figura 15: Equivalência de falhas

Na Figura 16(a) é mostrado o modelo BDD para o circuito mostrado na Figura 14. Na Figura 16(b) é mostrado o mesmo circuito usando o modelo SSBDD.

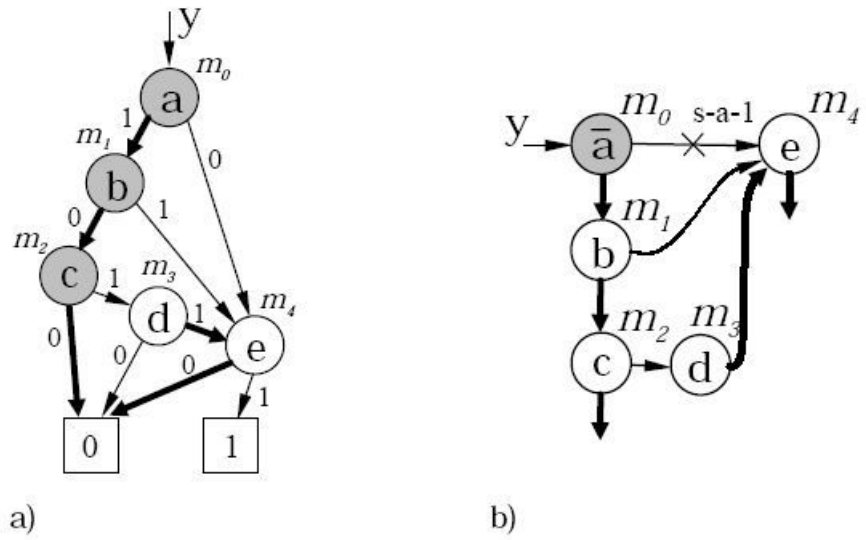


Figura 16: Representação de um circuito lógico usando o modelo BDD(a) e o SSBDD(b)
Fonte: (RAIK 2001, p. 45)

3.4 Método da Sensibilização do Caminho (*Path Sensitization Method)*

Inicialmente é necessário definir a álgebra ATPG. A álgebra ATPG é uma notação com o propósito de representar o circuito sem erros e o com erros, simultaneamente. Tem a vantagem de apenas necessitar uma propagação para os dois circuitos. Na Tabela 3 é mostrada a álgebra de Roth (ROTH 1966).

Tabela 3: Álgebra de Roth

Símbolo	Significado	Circuito Sem Falhas	Circuito Com Falhas
D	(1/0)	1	0
$\overline{D}$	(0/1)	0	1
0	(0/0)	0	0
1	(1/1)	1	1

Fonte: (BUSHNELL M. L.; AGRAWAL 2000, p. 160)

De acordo com a Tabela 3, a notação D significa que para um circuito sem falhas o sinal vale 1 e para um circuito com falhas o sinal vale 0. Já para $\overline{D}$ é 0 para um circuito sem falhas e 1 para um circuito com falhas. Maiores detalhes são descritos na seção 3.5.1.

O método da sensibilização do caminho é o mais usado (BUSHNELL M. L.; AGRAWAL 2000). Ele consiste em três passos:

- Ativação da falha: uma falha *stuck-at* é ativada forçando a linha de sinal num valor oposto ao valor da falha
- Propagação da falha: a falha é propagada por um ou mais caminhos para uma saída primária do circuito
- Justificação da linha: é necessário justificar uma linha através de suas entradas primárias para propagar a falha

No segundo e no terceiro passo, é possível acontecer um conflito, isso força ao ATPG a usar um *backtrack* e descartar um sinal e fazer um caminho alternativo. Considerando o exemplo da Figura 17, e uma falha s-a-0 em B. Para isso será feito a ativação da falha colocando B em 1. A propagação da falha é possível através de três cenários:

- Propagação através do caminho f- h- k- L

- Propagação através do caminho g - i - j - k - L
- Propagação simultânea pelos dois caminhos f - h - k - L e g - i -j -k - L

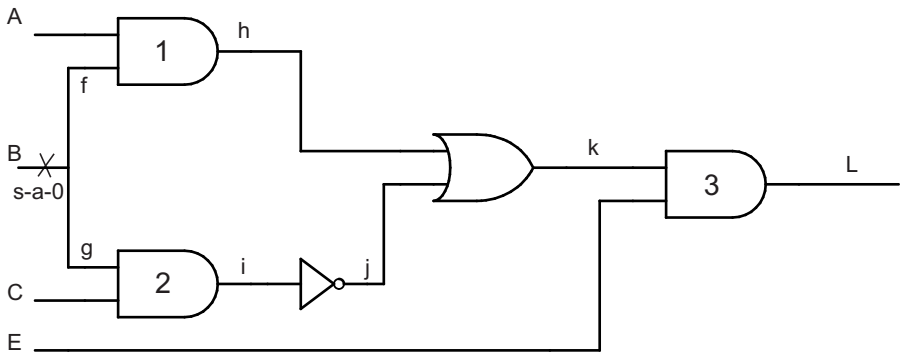


Figura 17: Exemplo de Path Sensitization Methods
Adaptado de (BUSHNELL M. L.; AGRAWAL 2000, p. 163)

Escolhendo o primeiro caminho para a propagação. Isso quer dizer que para todas as portas lógicas AND que estão no caminho(AND (1) e (3)) , que todas as suas entradas que não estão no caminho devem ser colocadas em 1 e similarmente para a porta lógica OR, todas as entradas que não estão no caminho devem ser colocadas para 0. Isso resulta em A = 1, j = 0 e E = 1. Agora é necessário justificar essas linhas internas. Para isso colocando i = 1 para justificar j = 0. A porta AND(2) deve ter sua saída em 1, porém ja possuí a entrada g = D. Quando temos numa AND uma entrada em D e a outra em 1 a saída será D, deste modo, não existe maneira de colocar o valor 1 na sua saída. Dessa forma é necessário usar outro caminho (*backtrack*).

Escolhendo o caminho 3, foi colocado os valores A = 1, C = 1 e E = 1 para assegurar a propagação. Dessa forma os valores de 1 e D na AND(2) terão sua saída i = D e a AND(1) em h = D. Como se tem uma inversora a saída j será $\overline{D}$. Como h = D e j = $\overline{D}$ na porta OR a sua saída será obrigatoriamente 1. Então o nosso sinal D não foi propagado para a saída e agora é necessário um *backtrack* e tentar o caminho 2.

Para propagar essa falha pelo caminho 2 é necessário colocar E = 1 e C = 1. Para isso o valor de i = D, j = $\overline{D}$, k = $\overline{D}$ e L = $\overline{D}$. Para isso é necessário justificar o valor de h com 0. Dessa forma para a porta lógica AND(1) ter sua saída h = 0 e com a entrada f = D é necessário colocar A = 0. Com isso conclui-se que o único teste possível para B s-a-0 é ABCE = 0111. Com esse vetor de entrada a saída L (L = $\overline{D}$) terá um valor L = 0 num circuito sem falha e L = 1 num circuito com falha.

3.5 Algoritmos ATPG

Os algoritmos ATPG possuem diversas funções, além da geração de padrões de teste. É possível também encontrar redundância ou sub-circuitos desnecessários. A definição da fronteira-D é comum a todos os algoritmos ATPG. A chamada fronteira-D são todas as portas lógicas com D ou $\overline{D}$ nas suas entradas e X nas suas saídas. A fronteira-D divide o circuito em duas partes, uma com as falhas e a outra sem (BUSHNELL M. L.; AGRAWAL 2000). Os algoritmos D, PODEM, GENETIC e RANDOM serão vistos nas próximas seções.

3.5.1 D-Algoritm

Na década de 60, testes estruturais foram introduzidos com o uso do modelo de falhas *stuck-at* (WANG L.T. ; WU 2006). Esse algoritmo foi proposto para se trabalhar com a geração de vetores de entrada em circuitos combinacionais (ROTH 1966). O algoritmo D utiliza os valores D e $\overline{D}$, que indicam 1/0 e 0/1 respectivamente e são propagados da linha onde existe a possibilidade do erro até uma saída do circuito. Quando existe um valor D em um ponto do circuito esse valor será 1 para um circuito sem erros e 0 para um circuito com erros. De uma forma análoga, pode-se dizer que para um valor $\overline{D}$ em um circuito sem erros esse valor será 0 e num circuito com erros será 1.

Tabela 4: Operadores Lógicos - AND

AND	0	1	x	D	$\overline{D}$
0	0	0	0	0	0
1	0	1	x	D	$\overline{D}$
x	0	x	x	x	x
D	0	D	x	D	0
$\overline{D}$	0	$\overline{D}$	x	0	$\overline{D}$

Tabela 5: Operadores Lógicos - OR

OR	0	1	x	D	$\overline{D}$
0	0	1	x	D	$\overline{D}$
1	1	1	1	1	1
x	x	1	x	x	x
D	D	1	x	D	1
$\overline{D}$	$\overline{D}$	1	x	1	$\overline{D}$

Fonte: (MOURAD S. ; ZORIAN 2000, p. 134)

Para provocar uma falha, é construído um cubo D primitivo por falha. Esse cubo é simplesmente os valores lógicos nas entradas e saídas de uma porta lógica, que provoca a falha em sua saída. Quando se deseja testar uma linha para um *stuck-at-1* é necessário colocar o valor contrário nessa linha[1], ou seja, valor 0. Nesse caso, o primeiro passo a se tomar seria escolher a porta lógica a ser testada, associado com a sua linha correspondente, e colocar o valor 0 nesse

[1]Teste de um *stuck-at-0*, colocar D. Teste de um *stuck-at-1*, colocar $\overline{D}$.

cubo. Então esse valor é trocado por $\overline{D}$. Se no caso a porta lógica for uma AND de três entradas e a falha é uma s-a-0 na saída, o cubo primitivo será [1,1,1,D]. É necessário que os três valores dessa porta lógica estejam em 1 para se propagar um nível 1 na saída dessa porta (esse 1 acaba sendo trocado por D). Esse cubo primitivo é ilustrado na Figura 18(a). Similarmente, para uma porta lógica NOR de duas entradas e uma falha s-a-1 na saída, é possível um dos três cubos [0,1,$\overline{D}$],[1,0,$\overline{D}$] ou [1,1,$\overline{D}$] desde que tenha um valor 1 em qualquer uma das entradas para colocar a saída em 0 (esse 0 acaba sendo trocado por $\overline{D}$, valor contrário a s-a-1, conforme mostrado na Figura 18(a) (MOURAD S. ; ZORIAN 2000). Já no caso de testar nessa NOR um s-a-0 na saída, o cubo será [0,0,D], conforme Figura 18a.

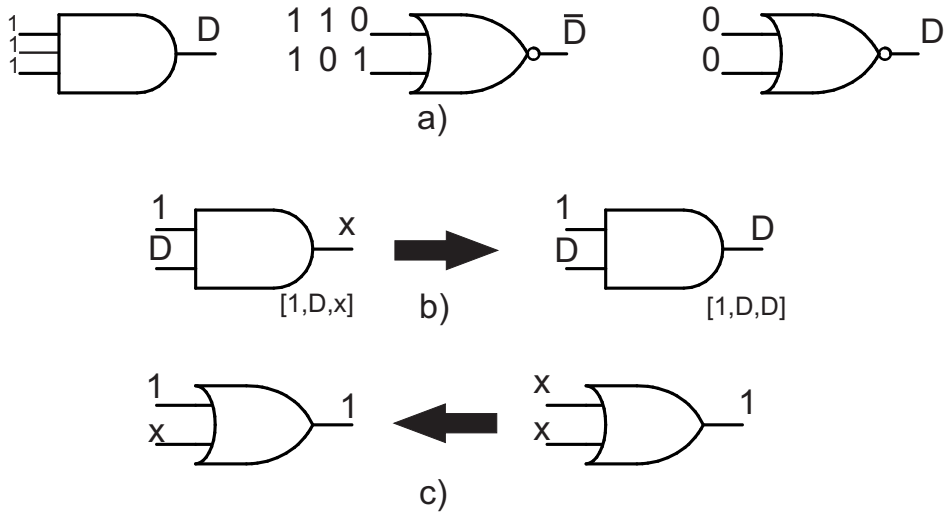


Figura 18: Operações do Algotimo D: (a) Cubo D primitivo (b) propagação do cubo D (c) justificação

O próximo passo seria enumerar as possíveis sensibilizações dos caminhos para uma saída primária. O cubo primitivo é propagado por qualquer um desses caminhos até a saída, usando as operações lógicas definidas nas Tabelas 4 e 5. Para propagar a falha na Figura 18(b) é formado o cubo D =[D,1,D]. Para isso são usados as operações mostradas na Tabela 4, ou seja, D AND 1 = D. Após feita a escolha do caminho, o algoritmo D começa tentando propagar o sinal de erro (D ou $\overline{D}$) da fonte de falha, até uma saída do circuito (WANG L.T. ; WU 2006).

O procedimento de escolha dos caminhos no algoritmo D não incorpora nenhum tipo de inteligência no processo de decisão. Dessa forma, às vezes é possível que alguns valores não sejam justificáveis no estado atual do circuito, porém devido a essa falta de inteligência, o algoritmo poderá necessitar de *backtrackings*[2].

[2]É quando um algoritmo qualquer tentar achar alguma solução para um problema, e determina candidatos para

3.5.2 PODEM

A IBM introduziu memórias DRAM em mainframes na década de 70 e para essas memórias o algoritmo D não conseguiu gerar os padrões de teste necessários, devido ao mecanismo de busca (BUSHNELL M. L.; AGRAWAL 2000). O algoritmo PODEM (*Path-Oriented Decision Making*) foi introduzido por (GOEL 1981) para trabalhar com esse tipo de circuitos. Foi o primeiro algoritmo ATPG que restringiu a sua procura às entradas primárias. Reduziu o tamanho da árvore de 2^n para 2^{PIs}, onde n é o número de portas lógicas e PIs é o número de entradas primárias. Essa restrição às entradas primárias do circuito mostrou-se uma eficiente técnica para reduzir a complexidade do algoritmo D, pois geralmente, o número de entradas do circuito é expressivamente menor que o total de portas do circuito (WANG L.T. ; WU 2006).

Uma subrotina chamada *X-PATH-CHECK* foi introduzida para checar se a fronteira-D ainda existe. Se não o algoritmo PODEM usa *backtracks* imediatamente. O algortimo PODEM toma decisões baseadas somente nas PIs. O algoritmo PODEM a cada busca do ATPG, verifica se a falha foi ativada. No momento em que a falha é ativada, o PODEM verifica se existe um caminho com valores indeterminados (valor X), iniciando em uma das portas que estão na fronteira-D e terminando em uma PO. O caminho que possui somente valores X é conhecido como caminho-X. No caso de não existir esse caminho, possivelmente, todos os caminhos estão sendo bloqueados pela lógica. Nesse caso o algoritmo tenta achar outro caminho-X, que seja, o melhor caminho, baseado na medida da observabilidade das portas que estão na fronteira-D.

Para a ativação da falha e propagação do erro, o algoritmo PODEM possui três procedimentos: *backtrace*, implicação e *backtracking*. A etapa de *backtrace* é desenvolvida excitando um valor lógico em uma PI do circuito a fim de conseguir justificar uma linha desejada do circuito. Para isso, o algoritmo excita valores na linha até uma PI. Após isso, o algortimo passa para a etapa de implicação. Nessa etapa, o algoritmo implica os valores lógicos nas saídas das portas do circuito, de acordo com os valores lógicos em suas entradas. No caso de um ou mais conflitos, ou seja, da impossibilidade de ativação da falha ou caso a fronteira-D se torne vazia, deverá ser retomada alguma decisão(*backtracking*).

Outro algoritmo muito conhecido é o FAN (*fan-out-oriented test generation algorithm*), que é um algoritmo mais rápido e mais eficiente que o algoritmo PODEM, de acordo com testes

essa solução e acaba abandonando um candidato x ("backtracks") devido a esse não conseguir uma possível solução.

em grandes circuitos combinacionais (FUJIWARA 1983). Esse algoritmo foge do escopo desse trabalho, por isso não foi descrito.

3.5.3 GENETIC

Algoritmos genéticos são uma família de modelos computacionais inspirados na evolução. O algoritmo genético modela uma solução para um problema específico em uma estrutura de dados como a de um vetor de teste e aplica operadores que conseguem recombinar estas estruturas, preservando certas informações. O algoritmo inicia uma população (geralmente randômica) de vetores de testes. Esses vetores são então avaliados para gerar oportunidades reprodutivas de forma que um vetor de teste que representa um solução mais eficaz tenha maiores chances de se reproduzir do que os que apresentarão uma solução insatisfatória. A definição de uma solução melhor ou pior é tipicamente relacionada à população atual (WHITLEY 1994).

De uma forma geral existem dois componentes do algoritmo genético que são dependentes do problema: a codificação do problema e a função de avaliação. Para um problema de otimização de parâmetros onde deve-se otimizar um conjunto de vetores de teste para se conseguir uma alta cobertura de testes. Para se conseguir isso, o procedimento funciona com uma seqüência de vetores de teste, chamados de população,que é melhorada iterativamente. Cada iteração é chamada de uma nova geração. Vetores de uma geração são produzidos pelos vetores da geração anterior, utilizando operações conhecidas como *crossover*, mutação e seleção. No *crossover* bits dos dois vetores da velha geração são combinados para construir 2 vetores da nova geração. Na mutação, bits de um vetor da velha geração são manipulados para criar um vetor da nova geração. Na seleção, 2 indivíduos são selecionados, onde existe a tendência em direção à seleção dos indivíduos altamente aptos. O *fitness* da nova geração é avaliado por simulação das características necessárias, tais como inicialização e detecção de falhas. A criação de vetores em gerações anteriores tem uma tendência ao maior *fitness* (BUSHNELL M. L.; AGRAWAL 2000).

3.5.4 RANDOM

Os vetores de testes também podem ser gerados aleatoriamente. O custo de gerar esse teste é mínimo. Um simulador de falhas é necessário para se testar a qualidade dos testes e a taxa

de cobertura. A vantagem do teste randômico é que consegue detectar uma alta porcentagem de falhas *stuck-at*. Conseqüentemente, muitas ferramentas comerciais ATPG usam o teste aleatório num primeiro estágio da geração dos vetores de testes e logo após aplicam testes determinísticos para lidar com as falhas resistentes (MOURAD S. ; ZORIAN 2000).

3.6 Conclusões

Nesse capítulo foi visto o conceito geral do modelo BDD. Uma classe especial de BDDs chamada de SSBDD foi explicada, a qual é mais adequada para a modelagem de um circuito digital do que o modelo BDD. Nesse trabalho foi usado somente o modelo SSBDD para se realizar a verificação de circuitos digitais. Os algoritmos PODEM, GENETIC e RANDOM foram usados na verificação e os resultados obtidos são discutidos no capítulo 6. Nas seções 4.2.2 e 4.3 é descrito com detalhes o processo para se verificar os circuitos digitais.

4 ARQUITETURA PROPOSTA (ARM E FPGA)

Com o crescimento da complexidade dos circuitos VLSI (*Very Large Scale Integrated Circuits*) e SOC (*System-on-a-Chip*), a geração de testes acabou se tornando uma das etapas mais complicadas e demoradas no domínio da concepção de circuitos integrados. Quanto mais complexos se tornam os sistemas eletrônicos, mais importante se tornam as etapas de verificação e teste.

A arquitetura proposta foi desenvolvida para verificação de circuitos digitais. Uma das grandes dificuldades de se conseguir usar um moderno ATE, está em seu alto custo. De acordo com Agrawal (BUSHNELL M. L.; AGRAWAL 2000), o custo de um ATE chega a alguns mil dólares por pino do componente que está sendo testado.

A arquitetura desenvolvida pode ser dividida tanto em verificação em software como em hardware. Na verificação em software, é possível com uma descrição de um circuito, usar os algoritmos ATPG para testá-lo. Foi desenvolvido um sistema computacional escrito na linguagem Java que trabalha em conjunto com as ferramentas Leonardo Spectrum e Turbo Tester para se realizar a verificação. Esse sistema computacional desenvolvido gerencia os outros programas para se facilitar a verificação de circuitos digitais e será visto nesse capítulo.

Da mesma forma, o sistema computacional desenvolvido em Java, também é usado para a verificação em hardware. Para isso foi desenvolvida uma plataforma de baixo custo que usa um microprocessador ARM e um FPGA. O ARM estudado nesse capítulo é usado para receber dados pela rede e transferir para o FPGA, que é usado para se emular o DUT.

4.1 Microcontroladores ARM

Uma tendência do mercado é termos os produtos com um baixo consumo de energia e alto poder de processamento, uma qualidade desejável em muitos produtos eletroeletrônicos. Com essa necessidade de mercado surgiram os microprocessadores ARM (*Advanced RISC Machine*). Já faz um certo período que os microprocessadores ARM de 32 bits são muito usados em PDA´s, celulares, videogames, etc. O núcleo (*core*) do ARM está disponível para diversos fabricantes como Cirrus, Philips, Analog Devices, etc (SOUZA 2006). A plataforma de verificação proposta nesse trabalho emprega uma linha de microprocessadores ARM9. Por causa dos seus cinco estágios de *pipeline*, o processador do ARM9 pode trabalhar com altas freqüências de *clock* se comparado à família do ARM7. Esses estágios extras aumentam o desempenho do processador. A memória foi redesenhada para a arquitetura *Harvard*, que separa os dados e o barramento de instruções. O primeiro processador da família ARM9 foi o ARM920T, que inclui de uma forma separada a memória *cache* e a MMU[1](*Memory Management Unit*). Esse processador pode ser usado por sistemas operacionais (SLOSS A.N.; SYMES 2004).

Nesse trabalho inicialmente foi utilizada a TS-7300 da Technologic Systems (TECHNOLOGIC). Essa placa possui um ARM920T(EP9302) e mais um FPGA[2] (Cyclone II), conforme mostrado na Figura 20. Essa plataforma foi escolhida devido a alta capacidade de processamento do ARM, por possuir um FPGA integrado para se emular o DUT e possuir uma porta de rede. O sistema computacional desenvolvido em Java envia dados de testes para esse ARM pela rede através de *socket*. O *socket* é descrito na seção 4.3.1. Primeiramente foram desenvolvidos programas em linguagem C para o ARM apresentado na Figura 19, para receber dados enviados pelo sistema computacional em Java por *socket* e para acessar os pinos de I/O do ARM.

Figura 19: Microcontrolador ARM9 - EP9302 da Cirrus Logic

[1]É um dispositivo de hardware que transforma endereços virtuais em endereços físicos.

[2]Hardware reconfigurável, o qual tem as suas funcionalidades definidas exclusivamente pelos usuários e não pelos fabricante.

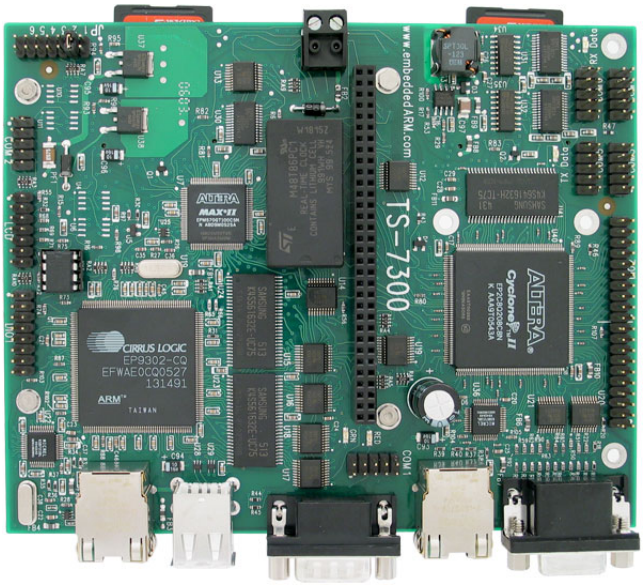


Figura 20: Placa TS-7300 da Technologic Systems

Após feita essa etapa o passo seguinte seria transformar esse FPGA em um DUT para ser testado pelo pinos de I/O do ARM, porém esse FPGA já vem com VGA[3] carregado nela e isto trouxe algumas complicações, pois, na tentativa de se reprogramá-la para funcionar somente como um DUT, o ARM acabava passando a um estado de *freeze*[4] , devido ao kernel que está no ARM tentar acessar a parte do VGA que foi apagado. Dessa forma, para resolver isso ou se reescrevia uma parte do kernel, o que seria inviável pelo tempo ou seria necessário colocar o DUT juntamente com o VGA implementado na Cyclone II. A descrição do VGA para essa placa é aberta no site *opencores*[5]. Porém, devido à limitação do tempo, acabou sendo mudada a plataforma usada. Com isso foi escolhida a placa TB500A da Tiny-Tech (TINY-TECH 2006) que possui o mesmo ARM920T RISC (EP9302) do fabricante Cirrus Logic conforme mostrado na Figura 21. Esse ARM trabalha na mesma freqüência do ARM da Technologic System, ou seja, de 200MHz. Uma grande diferença nessa plataforma da TB500A é que ela não possui um FPGA, agora sendo necessário mais uma placa para o FPGA.

A TB500A é uma plataforma de desenvolvimento de sistemas embarcados. Essa placa possui Ethernet 1/10/100Mbps, USB, serial, FLASH/SDRAM, SPI, AC97/I2S áudio e ADC. A memória FLASH de 64MB usada é dividida em duas seções. A primeira ocupa 2MB e é reservada para a imagem do *kernel*. A segunda é usada como partição usada pelo *root*. Uma imagem

[3]VGA (*Video Graphics Array*) refere-se ao hardware de vídeo introduzido pela IBM.

[4]Ocorre quando um programa de um microcomputador ou todo o sistema não responde mais às entradas de teclado e mouse.

[5]$http://www.opencores.org/projects.cgi/web/ts7300\_opencore/overview$

do *kernel* do Linux já vem instalada. O *kernel* usado é o Linux 2.6.x que é disponibilizado pela Cirrus Logic. O *boot loader* da placa é o TinyBOOT. A placa do ARM foi utilizada para enviar e receber dados do FPGA através dos seus pinos de entrada e saída de propósito geral (GPIO-*General Purpose Input/Output*). Para se acessar a placa do ARM, foi usada a interface serial para usar o microcomputador como um terminal, dessa forma configurando a serial com uma velocidade de 57600 bps, sendo possível acessar os programas que estão na placa. Inicialmente foi desenvolvido *drivers* em linguagem C para acessar os pinos de I/O. Com o *driver* desenvolvido foi feito programas em linguagem C para acessar esse *driver* e este acessar os pinos de I/O do ARM. Mais tarde foi descoberto que é possível acessar os registradores que dão acesso aos pinos de I/O sem a necessidade de usar *drivers*. A partir daí, foi simplificado o desenvolvimento dos programas.

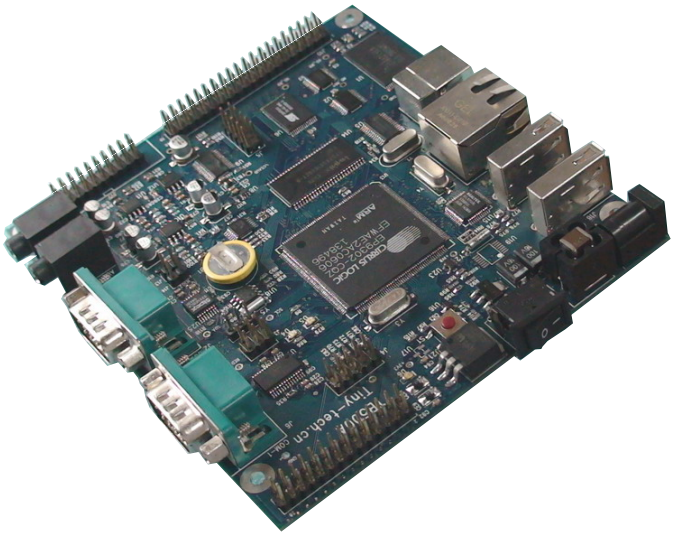


Figura 21: Placa TB500A da Tiny-Tech

A transferência dos arquivos para o ARM é descrita na seção 4.1.1. Essa transferência é necessária devido ao compilador ser executado no microcomputador. Na placa TS-7300 o compilador ficava na própria placa, o que facilitava bastante o desenvolvimento. A síntese para o FPGA é discutida na seção 4.2, onde este foi usado para se emular o hardware que está sendo testado, ou seja, o DUT.

4.1.1 Compartilhamento de arquivos usando NFS (*Network File System*)

Inicialmente a compilação para o ARM era feita no microcomputador e transferida por USB. Dessa forma, toda vez era necessário montar a USB no ARM, o que levava a um grande desperdício de tempo. Para se melhorar isso, começou a ser feita a transferência via conexão serial, mas mesmo assim existia um bom desperdício de tempo. Por isso acabou-se optando por usar o NFS. O NFS foi desenvolvido e implementado pela *Sun Microsystems* e endossado por grandes companhias. O NFS é um protocolo para se acessar arquivos remotamente (RUPPERT G.C.S. ;GEUS 2004). É o protocolo mais usado para se compartilhar arquivos em ambientes Unix. A idéia do NFS é realizar um acesso transparente de arquivos remotos, ou seja, as aplicações acessam arquivos que estão remotos utilizando a mesma semântica. Para isso o sistema de arquivos é montado em algum lugar da hierarquia local de arquivos e, a partir daí, é usado como um arquivo local. Para se fazer o compartilhamento, o servidor autentica o cliente apenas comparando o endereço IP da máquina que fez a requisição.

Na Figura 22 é mostrado um exemplo de compartilhamento usado entre o microcomputador (servidor) e o microcontrolador ARM (cliente). Esse compartilhamento foi necessário ser utilizado para facilitar a transferência de arquivos que foram compilados. O comando *ifconfig* é usado no ARM para configurar o IP. Logo após é usado o comando *mount*, que compartilha o diretório /home/testes do microcomputador sobre o IP 10.17.51.193 com o ARM no diretório /mnt/rede. Então toda a vez que for feito um programa em C para o ARM no microcomputador e compilado no diretório do mesmo, ele automaticamente, através de NFS, estará a disposição no diretório do ARM, não necessitando a transferência de arquivos.

```
ifconfig eth0 10.17.51.192
mount -o nolock 10.17.51.193:/home/testes /mnt/rede
```

Figura 22: Exemplo de compartilhamento de arquivos usando NFS

4.2 Arquitetura do FPGA

O FPGA (*Field Programmable Gate Array*) é um componente lógico que contém uma matriz bi-dimensional de células lógicas genéricas e *switches* programáveis. A estrutura conceitual

de um FPGA é mostrada na Figura 23. Uma célula lógica pode ser configurada para fazer uma simples função e o *switch* programável pode ser customizado para prover interconexão entre as células lógicas. Um circuito pode ser implementado especificando uma função para cada célula lógica e selecionando as conexões através do *switch* programável (CHU 2008). Uma das maiores utilizações do FPGA é na emulação de ASIC[6] (*Application Specific Integrated Circuit*). A validação da lógica do ASIC é um estágio crítico no processo de fabricação. A emulação implementa o circuito a ser fabricado em um FPGA, onde pode-se testar as funcionalidades desse circuito.

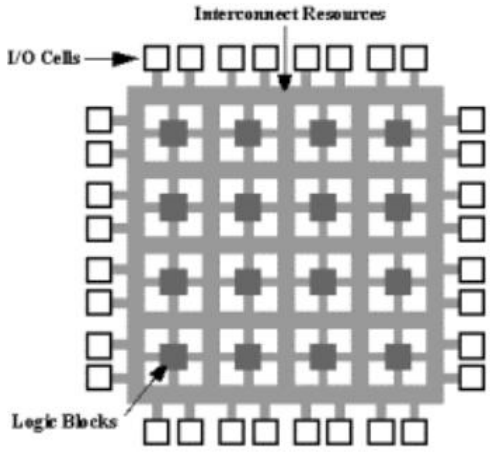


Figura 23: Estrutura conceitual de um FPGA

Existem 3 diferentes formas de se validar um ASIC:

- Protótipo em hardware. Consiste na implementação (protótipo) do circuito para análise e verificação das funcionalidades do circuito. Os experimentos são rapidamente executados pois a velocidade do protótipo é bem perto do circuito final

- Simulação em software. Essa técnica implementa um modelo funcional e/ou elétrico da implementação do circuito, que é simulado por um microcomputador. Uma das grandes vantagens deste método é a redução do custo, porém o tempo necessário para simular modelos complexos é muito alto, por isso em certos casos fica muito difícil implementá-la

- Emulação lógica. Ela consiste no uso do FPGA para implementar o modelo do circuito. Emuladores lógicos estão em algum lugar entre protótipos de hardware e simulação de software. O modelo do sistema emulado é muito mais rápido que uma simulação em software, porém não consegue ser tão rápido como o ASIC. Porém é muito mais fácil,

[6]Circuito integrado construído para executar uma tarefa específica.

barato e rápido, implementar o modelo num FPGA do que construir um protótipo. Também a emulação é mais flexível e fácil de usar. Por todas essas razões o FPGA é a mais usada para a validação de circuitos

A placa de desenvolvimento usada é uma Spartan3 da fabricante Digilent (DIGILENT 2006), conforme mostrado na Figura 24. A placa da Digilent possui 2Mbit de flash, 1Mbyte de SRAM, RS232, oscilador de 50MHz, VGA, PS/2 (teclado e mouse), JTAG (para gravação do bitstream), LEDS, *Switches*, display de 7 segmentos, botões, 3 portas de expansão, cada uma com 40 pinos[7].

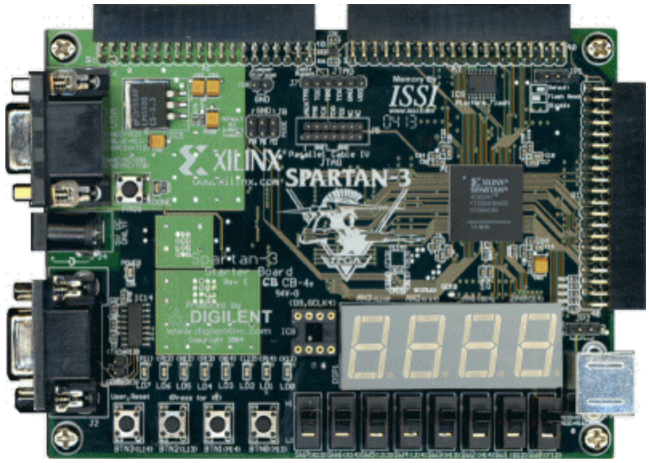


Figura 24: Placa da Digilent - Spartan 3

4.2.1 Fluxo de projeto em um FPGA

O fluxo de execução do modelo de um sistema em um FPGA inclui uma série de sucessivos processos que devem ser preenchidos. De acordo com Martinez (MARTINEZ 2007) este fluxo consiste nas seguintes etapas:

1. **Desenvolvimento do sistema** - Quando se desenvolve um novo sistema a partir do zero ou mesmo na construção de um sistema a partir de outros componentes, é necessário especificar um modelo funcional do sistema. Esse modelo, comumente especifica por meio de uma linguagem de descrição de hardware, tais com Verilog (IEEE 2005), VHDL (IEEE 2004) ou SystemC (IEEEa 2005), o ponto de partida de qualquer fluxo de desenvolvimento num FPGA

[7]Foi usado uma dessas portas para se realizar a comunicação entre ARM e FPGA

2. **Síntese** - A síntese lógica é um processo no qual o modelo do sistema é analisado para extrair a lógica dos elementos que foram especificadas no modelo e sua interligação

3. **Simulação funcional** - Uma vez que o modelo do sistema foi sintetizado, é muito aconselhável se simular o resultado da síntese para determinar se ela apresenta a funcionalidade pretendida. Vários simuladores estão disponíveis no mercado, sendo o ModelSim da Mentor Graphics um dos mais reconhecidos

4. **Mapeamento Lógico** - A lógica do mapeamento, mapeia o processo genérico da lógica dos elementos que foram extraídos pela síntese dos próprios recursos do FPGA selecionado

5. **Distribuição e roteamento** - Esta etapa inclui dois diferentes processos embora altamente interligados: distribuição e roteamento. O processo de distribuição consiste na seleção dos recursos da lógica livre do FPGA que será usado para implementar o circuito desejado. O processo do roteamento tenta executar as interconexões entre todos os elementos já colocados da lógica no FPGA

6. **Simulação temporal** - Antes de testar a implementação do sistema modelo sobre o real dispositivo, é aconselhável simular o resultado da distribuição e do roteamento. Este não é apenas uma simulação funcional, mas inclui a temporização e atrasos de toda a lógica e roteamento dos recursos do FPGA que foram utilizados na implementação do circuito

7. **Validação no FPGA** - O último passo é a verificação da implementação no FPGA. Uma correta execução sobre o FPGA concederá um provável sucesso do sistema

4.2.2 Síntese de circuitos HDL

Na seção 4.2.1 item 2, foi descrito brevemente a síntese de um circuito. No trabalho desenvolvido existem duas sínteses usadas. Existe uma síntese para ASIC conforme mostrado na seção 4.2.2.1, que foi usada para se simular os circuitos com ferramentas ATPG. A segunda síntese, foi desenvolvida para se executar os circuitos em um hardware real, ou seja, num FPGA conforme é apresentado na seção 4.2.2.2.

4.2.2.1 Síntese de circuitos para simulação

Para se testar um circuito é necessário termos um circuito desenvolvido em HDL. Com esse circuito será necessário converter com uma ferramenta de CAD[8] (*Computer-Aided Design*) para o padrão EDIF[9] (*Electronic Design Interchange Format*). Esse padrão EDIF gerado pelas ferramentas mais populares de CAD, incluindo Cadence, Synopsys, Mentor e Viewlogic é a conexão para o modelo SSBDD, que é usado para simular falhas. Os circuitos utilizados para verificação foram os *benchmarks* ISCAS85. Com a descrição do circuito é necessário validar antes de ser produzido o ASIC. O circuito c17 descrito em Verilog é mostrado na Figura 25. Neste circuito simples, como pode-se ver, as entradas são os sinais N1, N2, N3, N6 e N7 e as saídas são os sinais N22 e N23.

```
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5(N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
endmodule
```

Figura 25: Circuito C17 descrito em Verilog

Com essa descrição em *netlist* conforme visto na Figura 26 que é o arquivo EDIF, é possível passar para o modelo SBBDD conforme Figura 27. Os arquivos do modelo SBBDD possuem a extensão AGM. Com esses arquivos AGM, que são as descrições do circuito, é possível se utilizar as ferramentas de ATPG para se testar.

4.2.2.2 Síntese de circuitos para emulação

Da mesma forma que foi descrito na seção 4.2.2.1, para se validar o circuito usando emulação em um FPGA, é necessário ter-se um circuito descrito em HDL. Foi usado o ISE 9.1 fornecido pela empresa Xilinx, para a descrição do projeto em linguagem de descrição de hard-

[8]Desenho auxiliado por computador.

[9]É o formato predominantemente utilizado para guardar informações eletrônicas do *netlist* e do esquemático.

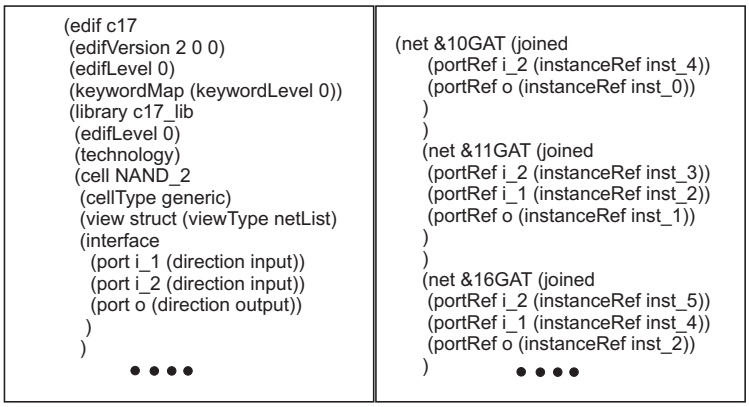


Figura 26: Circuito C17 descrito em EDIF

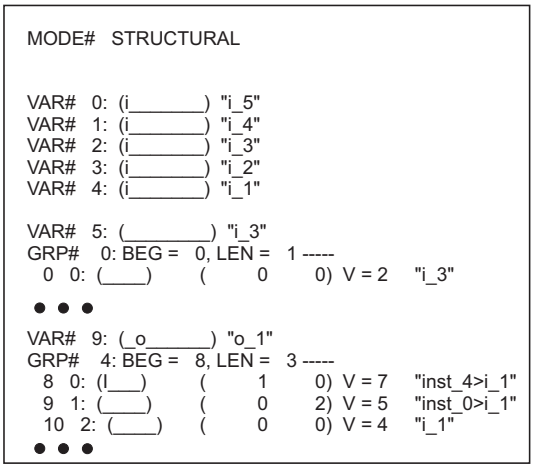


Figura 27: Circuito C17 descrito no modo estrutural (SSBDD)

ware (Verilog) e o mapeamento das entradas e saídas do sistema na placa FPGA para comunicação com o ARM. Posteriormente é realizada a síntese do circuito que será testado, ou seja, o FPGA irá se tornar o DUT. Nesse contexto a plataforma proposta irá trabalhar com um hardware para verificação de circuitos digitais de baixo custo. Para se conseguir isso o testador contará com um FPGA para emular o hardware e aumentar a velocidade de simulação. Esse FPGA irá emular um hardware enquanto um ARM será responsável por enviar informações para as entradas do circuito emulado no FPGA e analisar suas saídas. Na seção 4.3 é mostrada a arquitetura do DUT usada em emulação. Se o circuito não se comportar como o previsto ele possui erros de projeto e será necessário corrigi-lo. Se o resultado da emulação do hardware estiver correto, o circuito estará pronto para ser produzido.

4.3 Arquitetura de verificação de circuitos usando um sistema computacional em JAVA

A arquitetura foi desenvolvida para verificação de circuitos digitais. Pode-se basicamente dividi-la em duas partes: verificação em software e hardware, conforme mostrado na Figura 28. Inicialmente, para se testar um circuito é preciso tê-lo descrito em linguagem HDL (*Hardware Description Language*). Para a verificação foram usados os circuitos do *benchmark* ISCAS85. Com esse circuito descrito em HDL é possível usar o sistema computacional desenvolvido em Java para converter esse circuito em HDL para um *netlist*. Para isso o sistema acessa o programa Leonardo Spectrum (Mentor Graphics) e faz a síntese do circuito para ASIC e o transforma em um *netlist* do circuito. Essa descrição em *netlist* do circuito é chamada de arquivo EDIF. Para esse EDIF foi desenvolvida uma biblioteca que contém a descrição das portas lógicas primitivas do circuito. O sistema em Java acessa o programa Turbo Tester e mais a biblioteca das primitivas

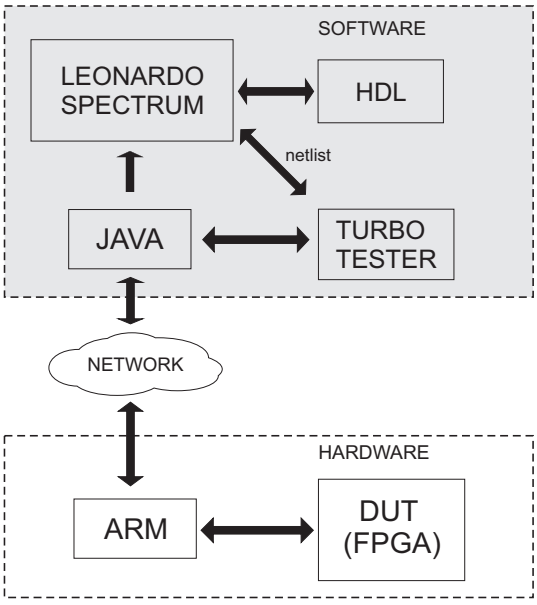


Figura 28: Arquitetura proposta

e faz a conversão do arquivo EDIF num arquivo SSBDD conforme mostrado na Figura 29. Esse arquivo é uma representação da lógica do circuito digital que será usado para a simulação do circuito. Com esses arquivos convertidos em SSBDD é possível usar as ferramentas de ATPG (Turbo Tester), conforme mostrado na Figura 30. Os algoritmos disponíveis são o PODEM, GENETIC e RANDOM. Com o uso dos algoritmos são injetados vetores de testes nas entradas dos circuitos e são monitoradas as saídas dos circuitos. Para acelerar essa simulação e também fazer uma análise real em hardware é possível usar um FPGA para emular o DUT, conforme

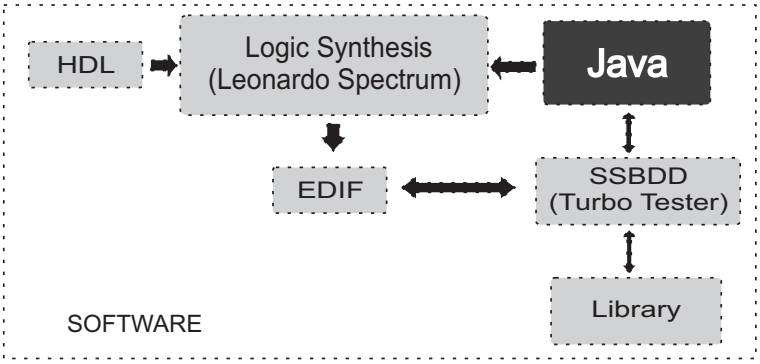


Figura 29: Arquitetura de verificação em software

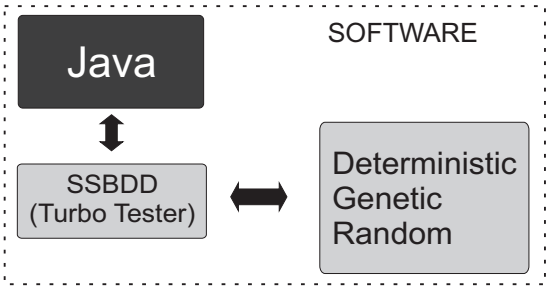


Figura 30: Arquitetura de verificação proposta

mostrado na Figura 28. O sistema em Java envia por *socket* os vetores de testes para o microcontrolador ARM. O ARM, recebendo esses dados, trata esses, deixando-os serializados e injetando no FPGA somente por um pino de I/O. Essa técnica de serializar os dados foi usada devido a limitação de pinos no ARM e de certos circuitos sob verificação possuírem mais entradas do que os pinos de I/O do ARM. No FPGA esses dados são recebidos e desserializados por um *shift register* conforme mostrado na Figura 32(a). Esse *shift register* é controlado por um *clock* do ARM. Após isso os dados são injetados no circuito e são recebidos por um outro *shift regiter* que serializa os dados novamente conforme mostrado na Figura 32(b). O ARM recebe esses dados e envia esses resultados da verificação dos circuitos por *socket* para o sistema em Java que compara essas respostas com respostas esperadas e classifica o circuito como bom ou ruim. Na Figura 31 é mostrada uma análise de dados entre o ARM e FPGA usando o analisador lógico.

Para se testar o DUT são usados vetores de testes na entrada e analisado suas saídas. Na Figura 32 o DUT que está sendo testado é o circuito c17 do *benchmark* ISCAS85. Dessa forma, para diferenciar vetores de entrada e saída, foi usado um padrão tanto em software como em hardware. Na entrada do DUT, um sinal em nível lógico alto é denominado de 1 e um baixo de 0. Já na saída do DUT, um nível lógico alto é mostrado como H e um baixo como L, conforme apresenta-se na Figura 33. Contudo, deve-se salientar que na emulação do circuito não existe

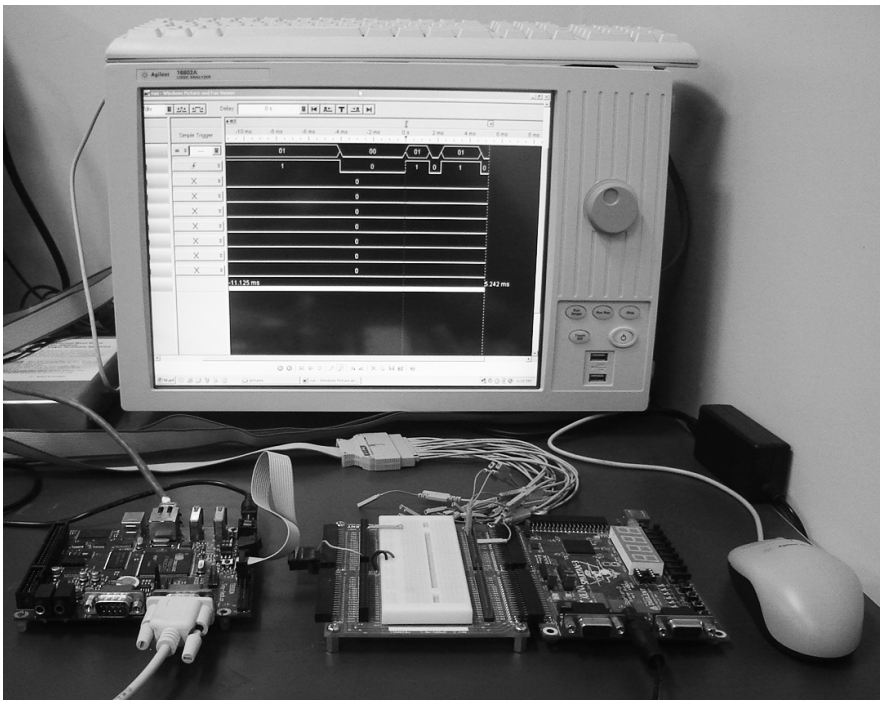


Figura 31: ARM, FPGA e analisador lógico

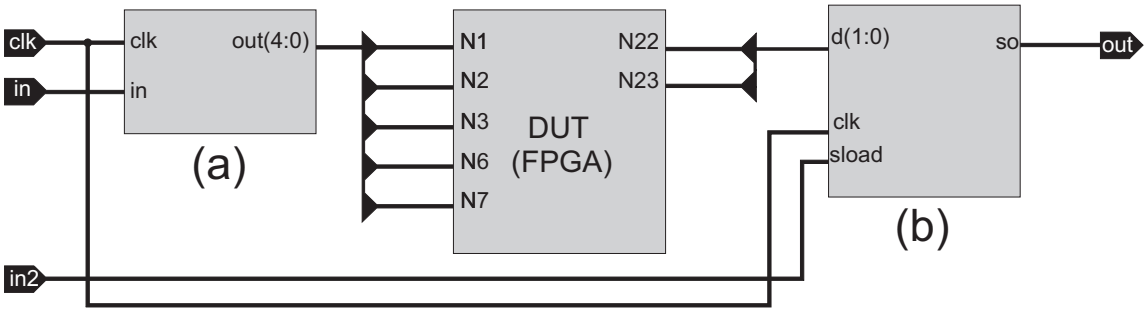


Figura 32: Arquitetura do DUT

sinal H e L, dessa forma o ARM converte os níveis lógicos na saída do FPGA em sinais H e L. Na Figura 33 é mostrada a verificação para o circuito c17. Esse circuito possui cinco portas de entrada e duas de saída.

4.3.1 Comunicação

A comunicação entre o sistema computacional desenvolvido em Java e o ARM inicialmente foi criada usando-se comunicação serial, com a biblioteca *RXTXcom* do Java. Contudo devido ao atraso gerado com a transmissão e recepção dos dados, a comunicação foi substituida por *sockets*. *Sockets* é um canal generalizado de comunicação entre processos. *Sockets* suporta comunicação entre processos independentes, e mesmo entre processos rodando em diferentes

Vetor	Vetores de Entrada	Vetores de Saída
1	11101	LH
2	00001	LL
3	10100	HL
4	11001	HL
5	01110	LL
6	00010	HH

Figura 33: Padrão usado para vetores de entrada e saída

máquinas se comunicando na rede. Quando é criado um *socket*, é necessário especificar o tipo de comunicação usada e o tipo de protocolo implementado (LOOSEMORE S. ; STALLMAN 1993). Para o *socket* implementado foram necessárias as seguintes informações:

- Endereço IP[10] (*Internet Protocol*) do servidor
- Porta onde se encontra o serviço no servidor
- Endereço IP do cliente
- Porta do cliente para a comunicação com o servidor

Quando se cria um *socket* entre programas, existem aqueles que aceitam conexões e aqueles que fazem as conexões. Um Servidor (*server*) é um programa que espera conexões e presumidamente provê algum serviço para outros programas. Ao contrário, um cliente (*client*) é um programa que se conecta a um servidor, usualmente para pedir que este lhe faça alguma coisa. Existem diferentes tipos de protocolos, que são usados em *socket*. Nesse trabalho foi usado o TCP. Além do protocolo é necessário informar o IP do *server* que especifica com qual computador você quer conversar. Também foi necessário informar a porta usada.

O presente trabalho usou *socket* conforme mostrado na Figura 34. O *socket* foi implementado usando a linguagem de programação Java no microcomputador (*server*) e em linguagem C no ARM (*client*). Foi definida a porta de comunicação 3490 para se estabelecer essa conexão.

[10]De uma forma genérica, pode ser considerado como um conjunto de números que representa o local de um determinado equipamento (normalmente computadores).

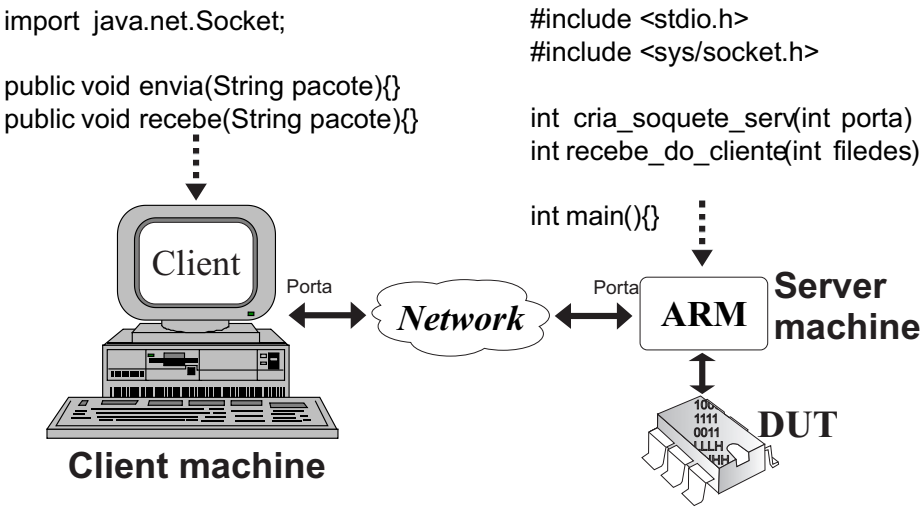


Figura 34: Comunicação usando *socket*

4.4 Sistema computacional em Java

Na Figura 35 é mostrada a ferramenta de verificação de circuitos desenvolvida em Java. Essa ferramenta controla a verificação dos circuitos. Nessa Figura são mostrado os arquivos possíveis a serem testados. A Tabela 6 mostra as extensões de alguns arquivos mostrados na

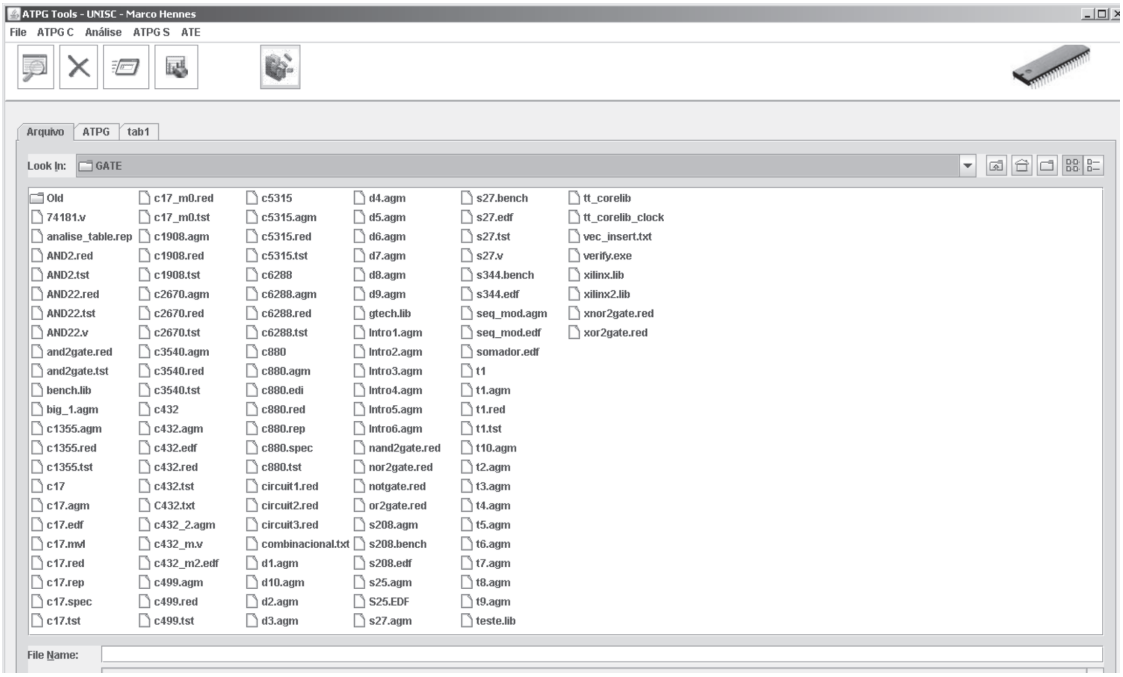


Figura 35: Ferramenta de verificação de circuitos digitais

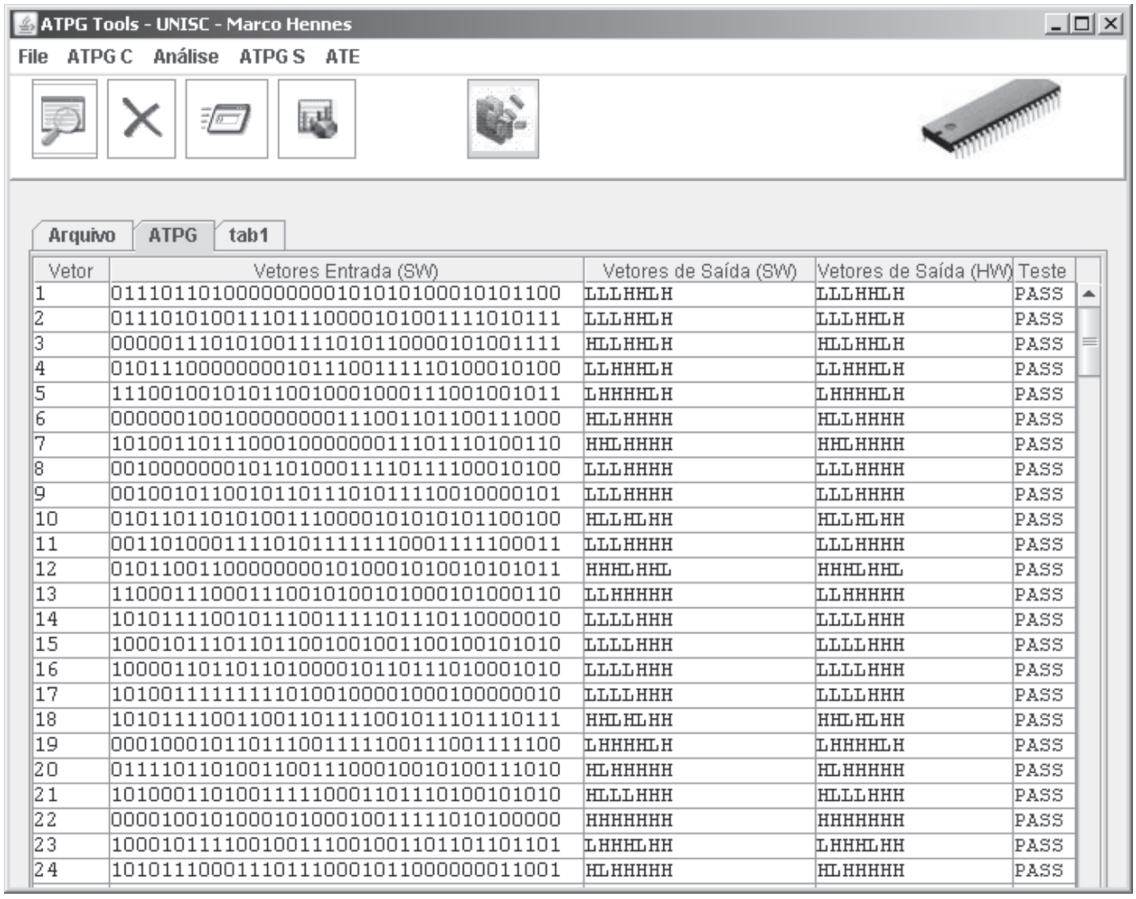
Figura 35 e a sua descrição.

A Figura 36 mostra uma análise de um circuito usando os vetores de testes na entrada e

Tabela 6: Descrição dos arquivos usados pela plataforma

Extensão do arquivo	Descrição do arquivo
EDF	Descrição do circuito no formato de *netlist*, gerado pelo Leonardo Spectrum
AGM	Descrição do circuito no formato estrutural, o qual será possível usar os algoritmos de simulação
TST	Vetores de testes
SEM EXTENSÃO	Análise de cobertura do circuito

analisando os vetores de saída. Para cada vetor é marcado a verificação como *PASS* ou *FAIL*, onde um *PASS* marca que esse circuito teve uma resposta correta para aquele vetor e um *FAIL* como incorreto.



Vetor	Vetores Entrada (SW)	Vetores de Saída (SW)	Vetores de Saída (HW)	Teste
1	01110110100000000010101010001010110 0	LLLHHLH	LLLHHLH	PASS
2	01110101001110111000010100111101011 1	LLLHHLH	LLLHHLH	PASS
3	00000111010100111101011000010100111 1	HLLHHLH	HLLHHLH	PASS
4	01011100000000010111001111101000101 00	LLHHHLH	LLHHHLH	PASS
5	11100100101011001000100011100100101 1	LHHHHLH	LHHHHLH	PASS
6	00000010010000000001110011011001110 00	HLLHHHH	HLLHHHH	PASS
7	10100110111000010000000111011101001 10	HHLHHHH	HHLHHHH	PASS
8	00100000001011010001111101111000101 00	LLLHHHH	LLLHHHH	PASS
9	00100101100101101110101111001000001 01	LLLHHHH	LLLHHHH	PASS
10	01011011010100111000010101010110010 0	HLLHLHH	HLLHLHH	PASS
11	00110100011110101111111100011111000 11	LLLHHHH	LLLHHHH	PASS
12	01011001100000000010100010100101010 11	HHHLHHL	HHHLHHL	PASS
13	11000111000111001010010100001010001 10	LLHHHHH	LLHHHHH	PASS
14	10101111001011100111111011101100000 10	LLLLHHH	LLLLHHH	PASS
15	10001011101101100100100110010010101 0	LLLLHHH	LLLLHHH	PASS
16	10000110110110100001011011101000101 0	LLLLHHH	LLLLHHH	PASS
17	10100111111111010010000100010000001 0	LLLLHHH	LLLLHHH	PASS
18	10101111001100110111100101110111011 1	HHLHLHH	HHLHLHH	PASS
19	00010001011011100111110011100111110 0	LHHHHLH	LHHHHLH	PASS
20	01111011010011001110001001010011101 0	HLHHHHH	HLHHHHH	PASS
21	10100011010011111000110111010010101 0	HLLLHHH	HLLLHHH	PASS
22	00001001010001010001001111101010000 0	HHHHHHH	HHHHHHH	PASS
23	10001011110010011100100110110110110 1	LHHHLHH	LHHHLHH	PASS
24	10101110001110111000101100000000110 01	HLHHHHH	HLHHHHH	PASS

Figura 36: Ferramenta de verificação de circuitos digitais

4.5 Análise em gráficos

A análise em gráficos criada para o sistema foi feita usando a biblioteca *JFreeChart*[11]. A ferramenta de verificação desenvolvida possui dois tipos de gráficos. O primeiro, que é o gráfico em barras é usado para a medição de tempo da geração dos vetores de teste. É possível comparar circuitos diferentes e também é possível usar algoritmos diferentes, conforme mostrado na Figura 37.

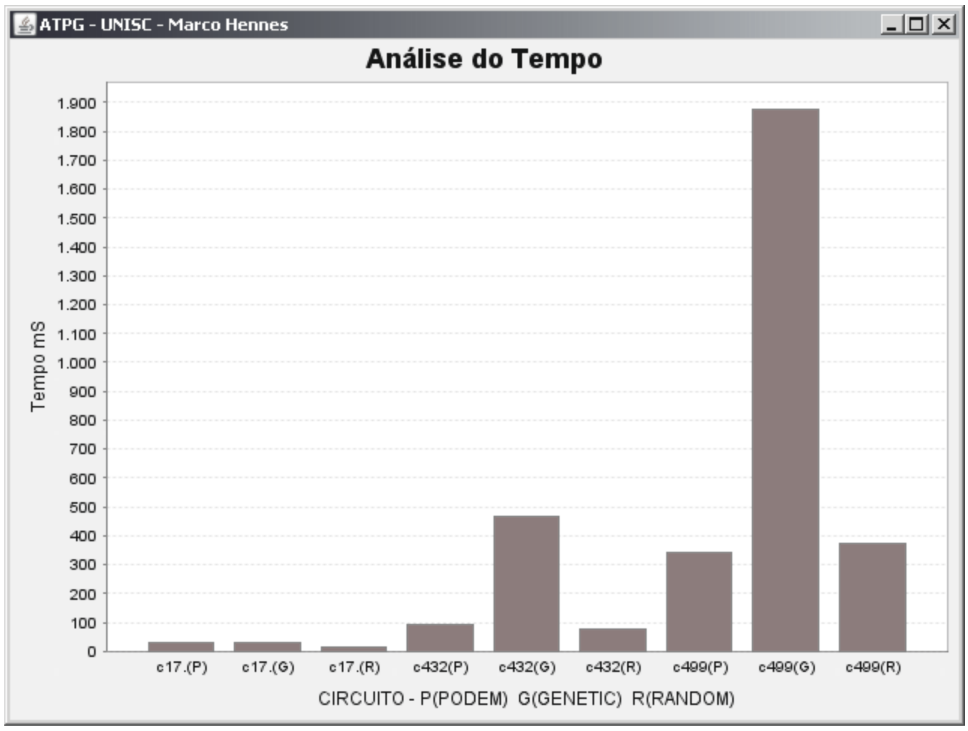


Figura 37: Análise de tempo

Nessa Figura é mostrada a verificação de 3 circuitos diferentes, onde a cada circuito foi aplicado um algoritmo diferente (PODEM, GENETIC, RANDOM). No circuito c17 quase não existe diferença de tempo, devido ao circuito ser muito pequeno. Já no circuito c432 e c499 é possível ver que o algoritmo GENETIC demora mais tempo que o PODEM e o RANDOM. Contudo, deve-se salientar que somente um baixo tempo não garante uma solução satisfatória, pois o ideal seria um baixo tempo e uma alta taxa de cobertura com uma quantidade reduzida de vetores de teste. Também é possível um segundo tipo de gráfico que faz a análise da cobertura desse circuito conforme a quantidade de vetores testados, apresentado na Figura 38. A cobertura da verificação da Figura 38 é calculada usando a fórmula 3.1. Nessa Figura é mostrado a

[11] *JFreeChart* é uma biblioteca gráfica para Java, que suporta uma ampla variedade de gráficos, incluindo gráficos (2D e 3D), barra de gráficos, etc.

verificação para o circuito c432 e são necessários um pouco mais de 75 vetores de testes para se conseguir uma taxa de cobertura de quase 95%. No Anexo A é apresentada a análise gráfica da simulação temporal dos circuitos do *benchmark* ISCAS85. Já no Anexo B, é mostrada uma análise da taxa de cobertura usando o algoritmo PODEM.

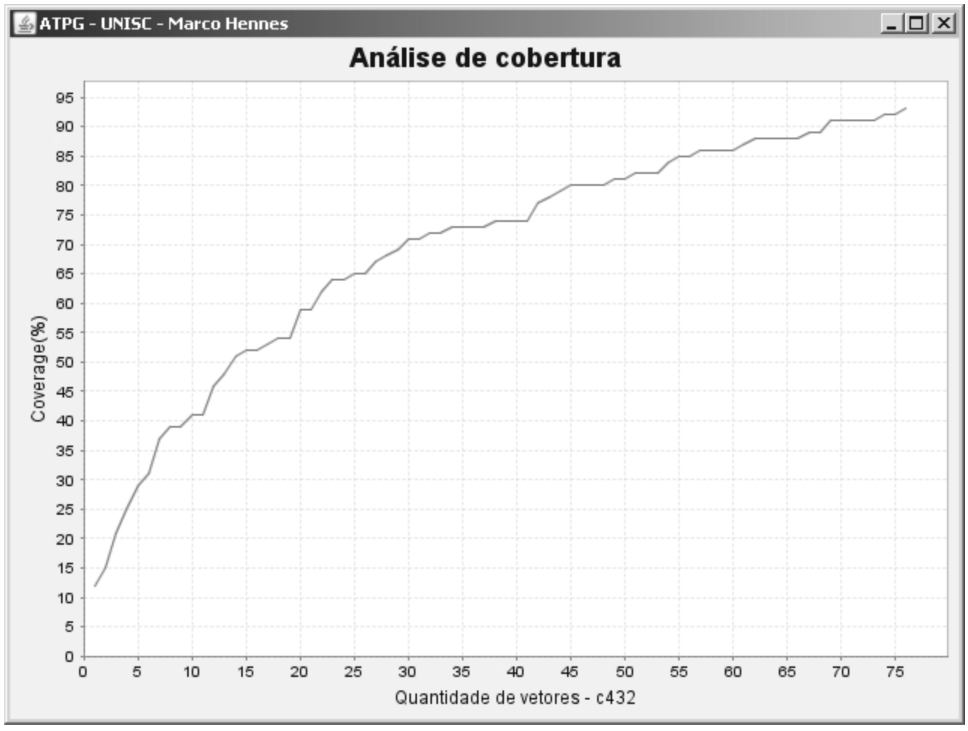


Figura 38: Análise de cobertura

4.6 Emulação de falhas

A utilização dos princípios da lógica de emulação e ferramentas no sentido de acelerar o modelo de experimentos baseado na injeção de falhas é conhecida como emulação de falhas. Uma das primeiras tentativas de utilização da lógica de emuladores com injeção de falhas foi proposto por (BURGUN et al. 1996) sob o nome emulação serial de falhas. Um programa foi usado para a implementação de modelos de sistemas para o FPGA e para gerar o arquivo de configuração dos circuitos *fault-free*. Este arquivo poderá ser usado para depurar o circuito e obter os valores esperados para os resultados considerados. Essa mesma ferramenta gerou um novo arquivo de configuração para cada falha a ser injetada no sistema. Estes arquivos reconfiguram a lógica emulada para emular o comportamento do sistema na presença de uma falha particular injetada. Neste trabalho, a injeção de falhas consistiu nos seguintes passos:

1. Execução do circuito livre de falhas em simulação
2. Configuração da lógica emulada para injetar a falha
3. Emulação do circuito defeituoso
4. Reconfiguração da lógica emulada para remover a falha

Assim, a metodologia da emulação serial de falhas envolve síntese e implementação de um modelo diferente para cada falha injetada no sistema, e a reconfiguração do FPGA para injetar e deletar cada falha encontrada. Embora o FPGA acelere a execução do processo de emulação de falhas, existe um enorme desperdício de tempo para injetar e deletar cada falha, pois se faz necessária uma nova síntese. Dessa forma, existem diversos métodos melhores para a injeção de falhas como o uso de registradores de deslocamento, multiplexadores e outros, descritos em (MARTINEZ 2007) (KOCAN e SAAB 2001) (ELLERVEE et al.) (KOCAN e SAAB 2007) (PELLEGRINI et al. 2008). Dessa forma, foi utilizado o método de injeção serial de falhas descrito acima onde é necessário reconfigurar o FPGA para cada falha inserida.

4.7 Conclusões

Nesse capítulo foi descrita a plataforma de verificação, tanto em simulação quanto em emulação. Essa plataforma trabalha com testes estruturais baseados no modelo de falhas *single stuck-at*. A plataforma foi desenvolvida e prototipada na placa TB500A da Tiny-tech, mais uma placa com um FPGA. O desenvolvimento de *drivers* e programas em C para a placa TB500A, necessitou de pesquisa, entendimento e escrita de rotinas para se acessar os pinos de I/O do ARM, para que fosse possível a comunicação entre ARM e FPGA. Também foi necessário utilizar o software ISE para a síntese dos circuitos que foram testados no FPGA.

A ferramenta desenvolvida em linguagem Java, permitiu a integração de várias ferramentas e o uso de gráficos para a análise dos circuitos e algoritmos utilizados.

5 ANÁLISE EM SIMULAÇÃO DO CIRCUITO C17 DO BENCHMARK ISCAS85

No capítulo 4, foi apresentada a arquitetura usada para verificação e no capítulo 6, serão apresentadas as verificações realizados nos circuitos. Essas verificações levam em conta somente os vetores que entram no circuito e os vetores que saem, dessa forma, não realizando uma análise mais profunda da parte interna do circuito. Este capítulo apresenta uma análise mais aprofundada levando em consideração apenas o circuito c17.

Esse estudo foi realizado para se obter uma análise mais detalhada do circuito e, no caso de o circuito ser defeituoso, tentar identificar onde está essa falha. Em circuitos maiores essa tarefa fica cada vez mais complexa, devido ao enorme número de portas lógicas presentes. Esse estudo foi separado em três seções, onde cada seção foi analisada em relação ao circuito c17, utilizando um algoritmo diferente.

5.1 Algoritmo PODEM

Nessa seção foi analisado o circuito c17 utilizando os vetores de teste gerados pelo algoritmo PODEM. O conjunto de verificação é constituído de 7 vetores e é descrito na Tabela 7.

Na Tabela 7 são mostrados os vetores de testes e suas respectivas respostas para circuitos sem defeitos. Cada vetor de teste consegue detectar uma quantidade determinada de falhas. Dessa forma, o vetor 1 mostrado na Tabela 7, consegue identificar 11 falhas, onde 7 delas são falhas s-a-0 e 4 são s-a-1. Cada número mostrado na Tabela 7, corresponde a um local de falha analisado no circuito c17 e é mostrado com um X, conforme é apresentado na Figura 39. Em cada ponto analisado é possível ter uma falha s-a-0 ou s-a-1. No caso do vetor 2, as falhas 12,13 (s-a-0) e 15 (s-a-1), estão riscadas na Tabela 7, devido ao vetor 1 já ter testado esses nós.

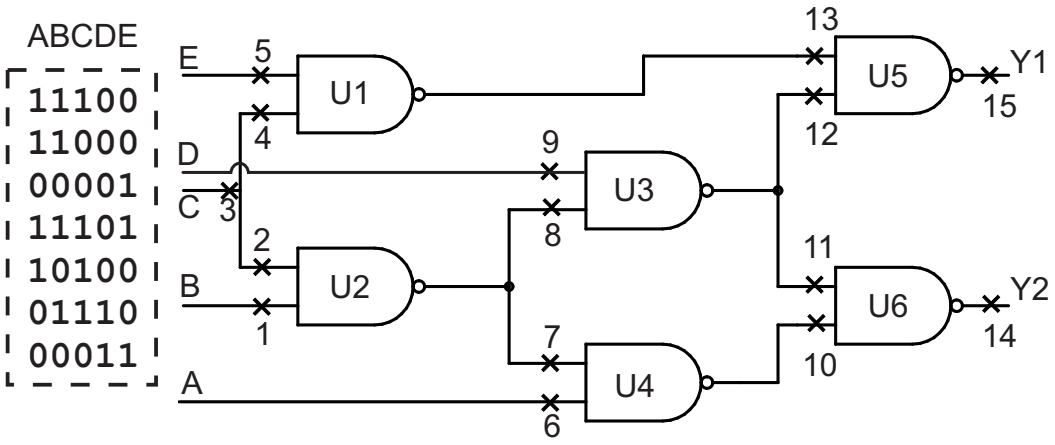


Figura 39: Circuito c17 do *benchmark* ISCAS85

Tabela 7: Vetores de teste para detecção de falhas do circuito c17 do *benchmark* ISCAS85

Vetor	Vetor de teste ABCDE	Saída Y1Y2	Quantidade de falhas detectadas	Local da falha (*Stuck-at*-0)	Local da falha (*Stuck-at*-1)
1	11100	LL	11	1,2,3,10,11,12,13	5,7,14,15
2	11000	HL	7	6,7,~~12,13~~,14	2,3,9,10,~~15~~
3	00001	LL	2	~~10,11,12,13~~	~~3~~,4,6,~~9,14,15~~
4	11101	LH	4	~~2,3~~,4,5,~~10,11~~,15	~~7~~,13,~~14~~
5	10100	HL	1	~~6,7,12,13,14~~	1,~~5,9,10,15~~
6	01110	LL	1	~~6,7~~,8,~~12,13,14~~	~~1,5,9,10,15~~
7	00011	HH	4	8,9,~~14,15~~	11,12

Conforme a seção 2.3.1, a quantidade de falhas em um circuito é descrita pela fórmula $2 * n$, onde n é o número de linhas do circuito. No circuito c17 foram encontrados 15 pontos para análise. Dessa forma, pela fórmula descrita acima, a quantidade de falhas é o dobro do número de linhas analisadas, ou seja, 30. A Figura 40 mostra a análise feita pelo sistema. Essa figura mostra os 7 vetores usados na verificação e sua taxa de cobertura. O vetor 1 mostrado na Tabela 7, consegue identificar 11 falhas *stuck-at*, dessas falhas, 7 são falhas *stuck-at-0* e 4 são *stuck-at-1*. O vetor 1 alcança uma taxa de cobertura 36.666667%. Essa taxa é calculada pela fórmula 3.1, descrita na seção 3.1. A taxa de cobertura é mensurada pela quantidade de falhas detectadas, no caso 11, pela quantidade de total de falhas, no caso de 30 (11/30).

Já o vetor 2 consegue identificar somente 7 falhas *stuck-at*, isso acontece devido ao vetor anterior já ter identificado as falhas 12, 13 (s-a-0) e 15 (s-a-1). Se no caso não tivesse sido usado o vetor 1, o vetor 2 teria identificado 10 falhas. Os outros vetores testados conseguirão aumentar pouco a taxa de cobertura, devido ao fato de já terem sido identificadas as falhas em diversos pontos, conforme mostrado na Tabela 7 e Figura 40. Dessa forma, o resultado de cobertura é

Coverage progress report:

Pattern 1: coverage 36.666667 % (11/30)
Pattern 2: coverage 60.000000 % (18/30)
Pattern 3: coverage 66.666667 % (20/30)
Pattern 4: coverage 80.000000 % (24/30)
Pattern 5: coverage 83.333333 % (25/30)
Pattern 6: coverage 86.666667 % (26/30)
Pattern 7: coverage 100.000000 % (30/30)

Figura 40: Análise da taxa de cobertura do circuito c17 feita pelo sistema

cumulativo e só pode ser alcançado se forem usados todos os vetores.

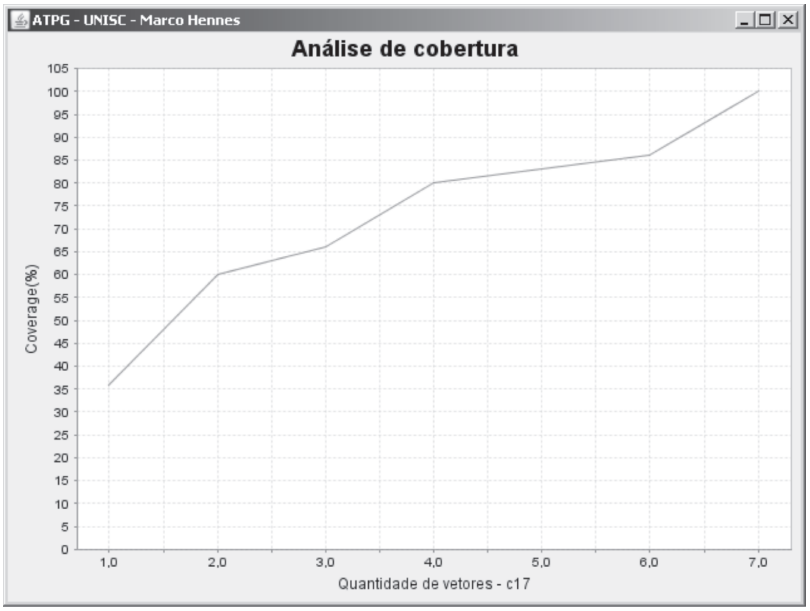


Figura 41: Análise gráfica da taxa de cobertura do circuito c17

Uma análise do circuito c17 para a falha localizada em 1 é apresentada na Figura 42. Dessa forma, o algoritmo tenta propagar essa falha na posição 1, até uma saída primária. Essa análise foi realizada para o vetor 1. Esse vetor coloca em B e C o valor lógico 1 e com esse valor é testado o valor contrátio, ou seja, 0 (*stuck-at-0*). Nessa figura é mostrada o valor lógico l e esse é trocado de 1 em B por D, conforme descrito na seção 3.4. Na porta lógica NAND U2, mostrado na Figura 42, uma entrada é D e a outra é 1. Para uma porta lógica NAND com entradas em D e 1, a saída será $\overline{D}$. Já para a porta U4, as entradas são 1 e $\overline{D}$ e sua saída ficara em D. Continuando a propagação do sinal, na porta U6 as entradas são D e 1, então a saída será $\overline{D}$. Se for analisado a Tabela 7 para o vetor 1, é visto que a saída deveria ser L, ou seja, valor em nível lógico 0. Então,se a saída for $\overline{D}$ e tiver valor 0, para esse vetor, o circuito não terá defeitos s-a-0 em 1. Já se o valor de $\overline{D}$ fosse 1, esse circuito teria falhas. Pela equivalência de falhas, quando se testa uma porta lógica NAND, é possível retirar falhas equivalentes na entrada 2 (s-a-0) e saída

7 (s-a-1) para a porta U2.

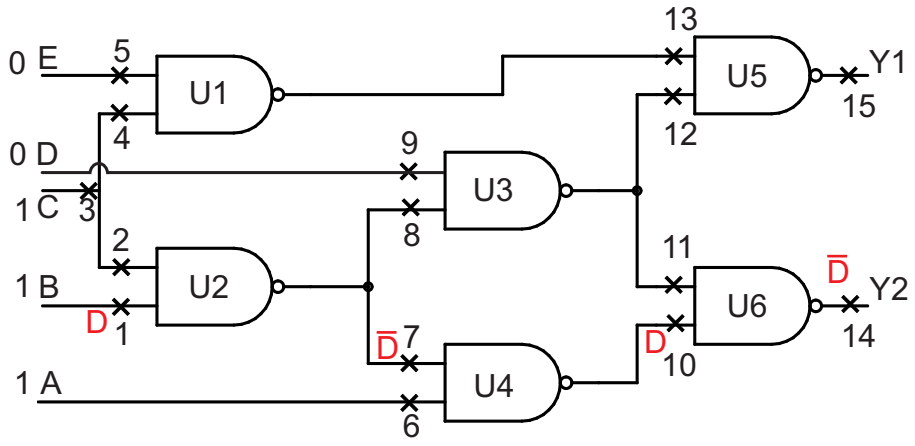


Figura 42: Análise do circuito c17 para a falha 1

5.2 Algoritmo GENETIC

O circuito testado nessa seção é o c17 e a Figura 43 foi mostrada novamente para um melhor entendimento e utilizou-se o algoritmo GENETIC. O conjunto dos 4 vetores testados é descrito na Tabela 8.

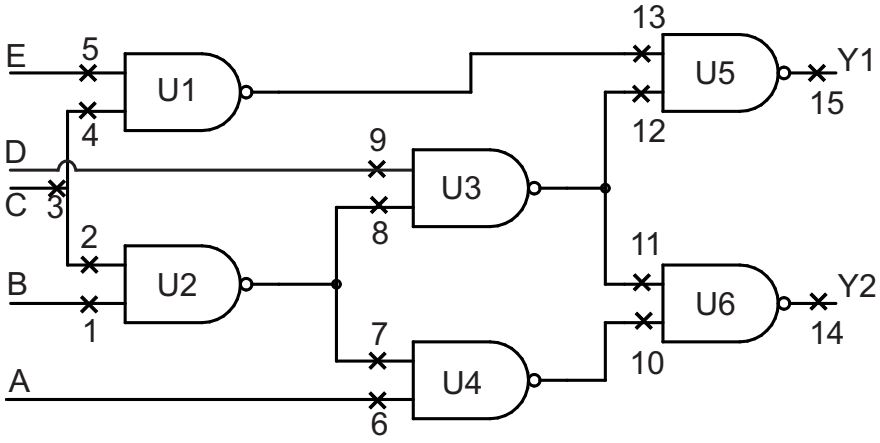


Figura 43: Circuito c17 do *benchmark* ISCAS85

O vetor 1 consegue detectar 12 falhas e consegue uma cobertura de 40%, conforme apresentado na Tabela 8 e Figura 44. Os outros vetores conseguem uma taxa de cobertura de 70%, 90% e 100%, respectivamente. É importante ressaltar que esse algoritmo conseguiu uma taxa de cobertura de 100%, conforme mostrado na Figura 45 e necessitou somente de 4 vetores de testes, enquanto o algoritmo PODEM necessitou de 7 vetores. O algoritmo GENETIC consegue esco-

Tabela 8: Vetores de teste para detecção de falhas do circuito c17 do *benchmark* ISCAS85

Vetor	Vetor de teste ABCDE	Saída Y1Y2	Quantidade de falhas detectadas	Local da falha (*Stuck-at*-0)	Local da falha (*Stuck-at*-1)
1	11110	LL	12	1,2,3,10,11,12,13	5,7,8,14,15
2	10101	HH	9	~~3~~,4,5,6,7,14,15	1,10,13
3	01011	HH	6	8,9,~~14,15~~	2,3,11,12
4	01001	LL	3	~~10,11,12,13~~	~~3~~,4,6,9,~~14,15~~

lher os melhores vetores de testes, ou seja, os que conseguem detectar o maior número possível de falhas, porém esse algoritmo necessita de um maior tempo computacional.

```
Coverage progress report:

Pattern 1: coverage 40.000000 % (12/30)
Pattern 2: coverage 70.000000 % (21/30)
Pattern 3: coverage 90.000000 % (27/30)
Pattern 4: coverage 100.000000 % (30/30))
```

Figura 44: Análise da taxa de cobertura do circuito c17 feita pelo sistema

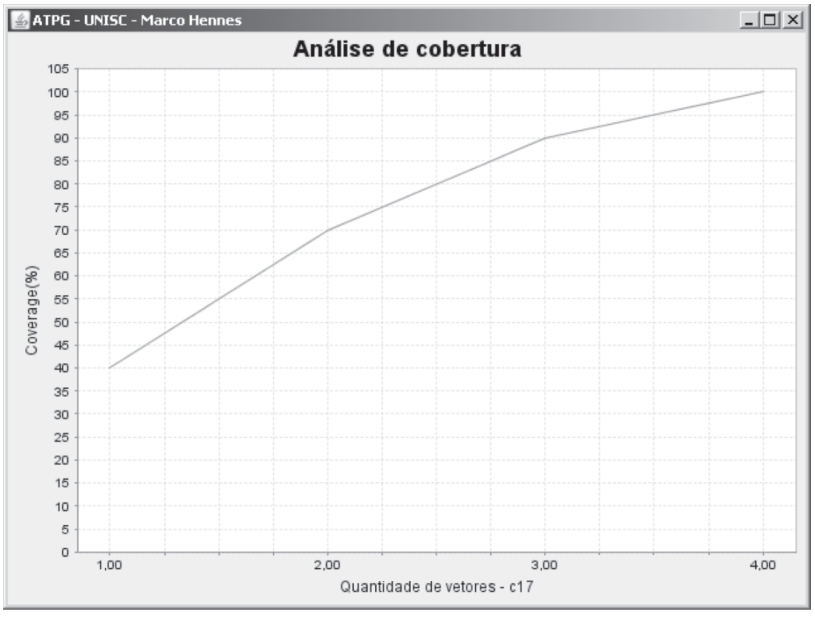


Figura 45: Análise gráfica da taxa de cobertura do circuito c17

5.3 Algoritmo RANDOM

O último estudo feito no circuito c17 foi usando o algoritmo RANDOM. A análise feita desse algoritmo é apresentada na Tabela 9. A análise dessa tabela mostra valores de testes similares ao algoritmo PODEM. Ainda são mostradas as taxas de cobertura do circuito nas Figuras

46 e 47.

Tabela 9: Vetores de teste para detecção de falhas do circuito c17 do *benchmark* ISCAS85

Vetor	Vetor de teste ABCDE	Saída Y1Y2	Quantidade de falhas detectadas	Local da falha (*Stuck-at*-0)	Local da falha (*Stuck-at*-1)
1	01110	LL	11	1,2,3,10,11,12,13	5,8,14,15
2	11001	HL	7	6,7,~~12,13~~,14	2,3,4,9,10,~~15~~
3	11100	LL	1	~~1,2,3,10,11,12,13~~	~~5~~,7,~~14,15~~
4	01111	LH	4	~~1,2,3~~,4,5,~~10,11~~,15	~~8~~,13,~~14~~
5	01001	LL	1	~~10,11,12,13~~	~~3,4~~,6,~~9~~
6	10101	HH	1	~~3,4,5,6,7,14,15~~	1,~~10,13~~
7	01011	HH	4	8,9,~~14,15~~	~~2,3~~,11,12

Coverage progress report:

Pattern 1: coverage 36.666667 % (11/30)
Pattern 2: coverage 63.333333 % (19/30)
Pattern 3: coverage 66.666667 % (20/30)
Pattern 4: coverage 80.000000 % (24/30)
Pattern 5: coverage 83.333333 % (25/30)
Pattern 6: coverage 86.666667 % (26/30)
Pattern 7: coverage 100.000000 % (30/30)

Figura 46: Análise da taxa de cobertura do circuito c17 feita pelo sistema

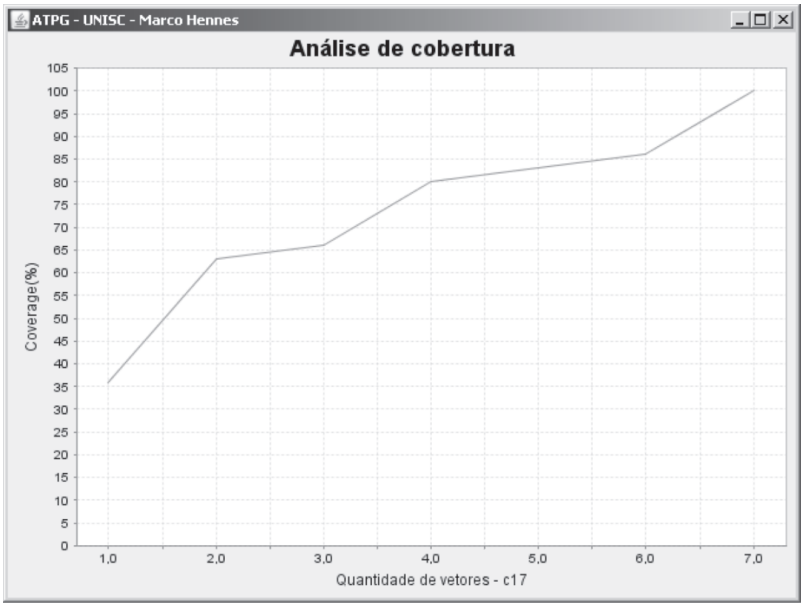


Figura 47: Análise gráfica da taxa de cobertura do circuito c17

5.4 Conclusões

Nesse capítulo foi analisado o circuito c17 usando os algoritmos PODEM, GENETIC e RANDOM. Essa análise foi realizada para se ter um melhor conhecimento do que acontece internamente no circuito e poder identificar o local da falha. A análise desse capítulo, onde se tenta localizar a falha e identificar as falhas equivalentes, foi realizada com sucesso, porém em circuitos maiores, deve-se ressaltar que existe um alto grau de complexidade.

6 RESULTADOS

6.1 Descrição dos circuitos testados

Nessa seção é investigado o tempo requerido para se verificar os circuitos em software e hardware. Para esta verificação, foram usados os circuitos do *benchmark* ISCAS85. No anexo C é mostrado uma descrição mais completa desses circuitos. Os circuitos que estão sendo verificados foram emulados num FPGA, sendo necessário o uso do programa ISE 9.1 (Xilinx) para a síntese dos circuitos. O circuito sintetizado é implementado numa Spartan 3 (XILINX 2004) da Xilinx. Já o ARM usado no hardware para injetar os sinais no FPGA é o EP9302 (CIRRUS LOGIC 2005) da Cirrus Logic.

A descrição dos circuitos do *benchmark* ISCAS85 é tabulada na Tabela 10. Nessa Tabela, são apresentadas as entradas e saídas (I/O), e o número de portas lógicas (Núm. Gates) e número de testes para os algoritmos PODEM, GENETIC e RANDOM dos circuitos, são tabulados nessa ordem. Deve-se salientar que a quantidade de vetores de testes gerados pelo algoritmo GENETIC pode variar um pouco, pois para este algoritmo não existe uma solução única.

6.2 Resultados Experimentais em Simulação

Na Tabela 11 é mostrada a análise em software para os circuitos do *benchmark* ISCAS85. Nessa Tabela, são mostrados os circuitos e seus respectivos tempos (T) para simulação e a taxa de cobertura (TC), conforme o algoritmo usado. Esse tempo leva em conta o tempo do sistema computacional desenvolvido em Java para se acessar o Turbo Tester, mais o tempo da ferramenta Turbo Tester para se analisar o circuito. Dessa forma, a cobertura será a mesma alcançada pela ferramenta Turbo Tester, porém o tempo será a soma dos tempos das duas ferramentas.

Tabela 10: Descrição dos *benchmarks* ISCAS85

	Núm. de I/O	Núm. de Gates	Núm. de Testes PODEM	Núm. de Testes GENETIC	Núm. de Testes RANDOM
c17	5/2	6	7	4	7
c432	36/7	160	72	48	76
c499	41/32	202	123	86	100
c880	60/26	383	84	56	92
c1355	41/32	546	118	118	118
c1908	33/25	880	149	123	185
c2670	233/140	1269	152	152	152
c3540	50/22	1669	201	158	250
c5315	178/123	2307	147	119	203
c6288	32/32	2416	41	23	47

Analisando a Tabela 10, sem levar em conta a taxa de cobertura, é visto que o algortimo que consegue o menor número de vetores é o GENETIC, seguido pelo PODEM e o RANDOM. Porém, com uma taxa de cobertura um pouco superior ao PODEM, o algoritmo GENETIC precisa de um tempo computacional maior que o PODEM, conforme apresentado na Tabela 11. Dessa forma, apesar do algoritmo GENETIC conseguir um número reduzido de vetores de testes, dependendo do circuito ele demanda um alto tempo de processamento. O algoritmo PODEM esta baseado no algoritmo D, ou seja, é uma melhoria do algoritmo D. O método de busca do algoritmo D, está baseado na quantidade de portas lógicas. Dessa forma, se houver um circuito com muitas portas lógicas, essa verificação será mais demorada. Já o método de busca do algoritmo PODEM está baseado no número de entradas desse circuito. Por isso quanto se aplicou a verificação para circuitos com muitas portas lógicas que possuem poucas entradas, como o circuito c1908, o algoritmo PODEM mostrou um desempenho superior em relação aos algoritmos GENETIC e RANDOM.

6.3 Resultados Experimentais em Emulação

Na tabela 12 são mostrados os resultados para a emulação em hardware. Os vetores que são usados na verificação em hardware, são os mesmos usados em software e são transferidos por *socket* pela ferramenta desenvolvida em Java. Dessa forma, a parte da verificação em hardware não possui o gerador de vetores de testes e a análise das saídas dos circuitos emulados é feita no sistema computacional. Como em hardware não existe o gerador de vetores de teste, a diferença de tempo na verificação usando diferentes algoritmos em emulação, esta relacionada a quantidade

Tabela 11: Taxa de cobertura e tempo de simulação para o circuito

Circuito	PODEM		GENETIC		RANDOM	
	T(ms)	TC	T(ms)	TC	T(ms)	TC
c17	15	100%	21	100%	16	100%
c432	47	86,20%	453	93,01%	78	93,01%
c499	172	99,63%	1860	99,63%	375	99,63%
c880	78	100%	1265	100%	329	99.39%
c1355	16	99,63%	16	99,63%	15	99,63%
c1908	234	99,53%	2579	99,60%	750	99,56%
c2670	16	95,01%	16	95,01%	16	95,01%
c3540	516	95,33%	7390	95,68%	1718	95,59%
c5315	750	98,98%	9703	99,29%	1516	99,29%
c6288	235	99,30%	8524	99,30%	1187	99,30%

Tabela 12: Emulação em Hardware

Circuito	Síntese para FPGA(min/s)	PODEM Tempo(ms)	GENETIC Tempo(ms)	RANDOM Tempo(ms)
c17	00:56	>0	>0	>0
c432	00:57	31	31	31
c499	00:59	78	109	109
c880	01:02	78	63	79
c1355	00:57	109	109	110
c1908	01:06	171	78	125
c2670	02:06	180	152	160
c3540	01:20	125	94	188
c5315	02:36	175	145	165
c6288	03:21	63	62	62

de vetores de teste usada. É possível notar que existe um menor tempo conforme apresenta-se na Tabela 12, se comparada aos mesmos circuitos e algoritmos mostrados na Tabela 11. Isso se deve ao fato da verificação em hardware ser mais rápida do que a em software, apesar de a parte em hardware não possuir o gerador de vetores. Porém, deve-se salientar que esse tempo visto na Tabela 12 leva em conta o tempo de envio e recebimento de dados por *socket*, o de tratamento dos dados na entrada e saída do DUT e o de propagação no DUT.

7 CONCLUSÕES

Nesse trabalho foi proposta uma metodologia de verificação de circuitos integrados. O objetivo do teste de circuitos integrados é identificar e isolar dispositivos falhos. Então, com o crescimento da microeletrônica no país, se faz necessário a criação e aperfeiçoamento de novas técnicas, para que sejam criados dispositivos de testes cada vez mais eficientes e de preferência com custos acessíveis às empresas de menor porte.

O estudo apresentado no início do presente trabalho inclui a descrição dos conceitos básicos na área de testes e sua aplicação para geração de testes usando o modelo de falhas *stuck-at*. Para isso, se fez necessário a descrição do circuito em um modelo matemático. O modelo matemático usado para representar os circuitos que foram testados, foi o SSBDD. Com o modelo de falhas e o modelo matemático, trabalhou-se com a verificação usando os algoritmos PODEM, GENETIC e RANDOM. Na seção 4, foi mostrada a arquitetura usada para a verificação de circuitos integrados. Nessa seção, foi usada a ferramenta Turbo Tester e Leonardo Spectrum, gerenciadas pela plataforma computacional desenvolvida em Java. Essa plataforma foi usada tanto para verificação em simulação, como emulação. Na parte da verificação em emulação, foi descrita a arquitetura do ARM e FPGA usadas. Já na seção 5, foi mostrado um estudo mais completo dos testes na parte interna do circuito c17. E na seção 6, são mostrados os resultados alcançados, usando a plataforma gerenciadora em Java.

Inicialmente a plataforma de verificação foi criada para trabalhar em placas de desenvolvimento (DIGILENT ,TECHNOLOGIC e TINY-TECH). O passo seguinte seria transformar essa plataforma de verificação em um ambiente usando um ARM e FPGA projetados dentro das dependências da UNISC. Dessa forma, o projeto da placa de circuito impresso ficou completo, na descrição dos circuitos (esquemático[1]) e na ligação dos componentes elétricos (roteamento[2]). No

[1] É uma representação gráfica das ligações elétricas de um circuito.
[2] É o processo que conecta os componentes eletrônicos através de trilhas.

momento essa placa projetada está sendo fabricada e necessitando posteriormente de montagem e testes. No Anexo E é mostrado o projeto da placa de circuito impresso. É possível num trabalho futuro o uso desse hardware para testes de circuitos, porém não sendo o objetivo desse trabalho.

Na seção 6, foram mostrados os resultados alcançados pela plataforma. Apesar de existir um certo atraso na propagação dos vetores de teste pela rede na verificação em emulação, conclui-se que a realização dessa verificação foi realizada com êxito. Em simulação a verificação também foi realizada de uma forma satistatória. Então, pelo que foi descrito, conclui-se que foi alcançada a metodologia de verificação proposta como contribuição. Já a parte do projeto de um testador de baixo custo usando uma linguagem de programação foi concluída usando duas plataformas de desenvolvimento (DIGILENT e TINY-TECH) de baixo custo e não com um projeto de um hardware próprio. Uso de um microcontrolador novo que possui um sistema operacional, foi concluído com êxito, pois esse é necessário para a emulação. Finalmente, um grande diferencial da plataforma desenvolvida é a sua flexibilidade. O ambiente de verificação está sendo executado no microcomputador, porém o ambiente de simulação poderia ser todo executado dentro do ARM. Como o ARM possui um sistema operacional Linux, seria possível rodar aplicações de verificação no próprio ARM. Também seria possível expandir o uso do ARM para aplicações em outras áreas de pesquisa.

Esse trabalho apresentou uma nova plataforma de verificação para circuitos combinacionais e foi publicado no XV Iberchip conforme mostrado no Anexo D. Com o sistema computacional desenvolvido em Java é possível se verificar circuitos usando diferentes algoritmos. Essa plataforma mostrou um ambiente de simulação e emulação em hardware utilizando FPGA para acelerar a verificação de circuitos integrados. Um grande diferencial dessa plataforma é que ela utiliza somente componentes de baixo custo. Outro grande diferencial é a integração de várias ferramentas, para verificação em software e em hardware.

7.1 Trabalhos Futuros

A plataforma desenvolvida, mesmo ficando muito longe de um moderno ATE, usa somente componentes de baixo custo, o que lhe proporciona um grande diferencial e abre caminho para que sejam elaborados outros trabalhos para se chegar mais perto dos ATE disponíveis no mercado. Como trabalho futuro pretende-se implementar essa plataforma para verificação de

circuitos seqüênciais. Também é possível implementar um método mais eficaz para se injetar e deletar as falhas, como o uso de registradores de deslocamento.

REFERÊNCIAS

[AL-ASAAD 1998]AL-ASAAD, H. *Lifetime Validation of Digital Systems via Fault Modeling and Test Generation*. Tese (Doutorado) — The University of Michigan, 1998.

[BERGER I.; KOHAVI 1973]BERGER I.; KOHAVI, Z. Fault Detection in Fanout-Free Combinational Networks. *Transactions on Computers*, v. 100, n. 22, p. 908–914, 1973.

[BRYANT 1986]BRYANT, R. Graph-based algorithms for boolean functions. *IEEE Trans. on Computers, C-35 (8)*, p. 677–691, 1986.

[BURGUN et al. 1996]BURGUN, L. et al. Serial fault emulation. In: *Annual ACM IEEE Design Automation Conference: Proceedings of the 33 rd annual conference on Design automation*. [S.l.: s.n.], 1996. v. 3, p. 801–806.

[BUSHNELL M. L.; AGRAWAL 2000]BUSHNELL M. L.; AGRAWAL, V. D. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits. Dordrecht*. [S.l.]: The Netherlands: Kluwer Academic Publishers, 2000.

[CHU 2008]CHU, P. P. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. [S.l.]: Wiley-Interscience, 2008.

[CILETTI 2002]CILETTI, M. *Advanced digital design with the Verilog HDL*. [S.l.]: Prentice Hall, 2002.

[CIRRUS LOGIC 2005]CIRRUS LOGIC. *EP9302 Data Sheet*. [S.l.], 2005.

[DIGILENT 2006]DIGILENT. *Spartan-3 Starter Kit Board User Guide*. [S.l.], 2006.

[ELLERVEE et al.]ELLERVEE, P. et al. Fault Emulation on FPGA: A Feasibility Study. In: *Proc. 21st Norchip Conference*. [S.l.: s.n.]. p. 92–95.

[FERREIRA 1998]FERREIRA, J. *Introdução ao projecto com sistemas digistais e microcontroladores*. [S.l.]: FEUP Edições, 1998.

[FUJIWARA 1983]FUJIWARA, H. . S. T. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, v. 32, p. 1137–1144, 1983.

[GIZOPOULOS D.; PASCHALIS 2004]GIZOPOULOS D.; PASCHALIS, A. Z. Y. *Embedded Processor-Based Self-Test*. [S.l.]: Dordrecht, The Netherlands: Kluwer Academic Publishers, 2004.

[GOEL 1981]GOEL, P. An implicit enumeration algorithm to generate tests for combinational circuits. *IEEE Transactions on Computers*, v. 30, n. 3, p. 215–222, 1981.

[HANSEN et al. 1999]HANSEN, M. et al. Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering. *IEEE DESIGN & TEST OF COMPUTERS*, IEEE Computer Society, p. 72–80, 1999.

[HENNES e SANTOS R.R.; MOLZ 2009]HENNES, M.; SANTOS R.R.; MOLZ, R. Plataforma para testes de circuitos digitais. XV Workshop Iberchip, p. 439–443, 2009.

[IEEE 2004]IEEE. *Behavioural languagesPart 1: VHDL language reference manual. IEEE Standard No. 61691-1-1*. [S.l.], 2004.

[IEEE 2005]IEEE. *Verilog Hardware Description Language.IEEE Standard No. 1364*. [S.l.], 2005.

[IEEEa 2005]IEEEa. *SystemC Language Reference Manual. IEEE Standard No.1666*. [S.l.], 2005.

[JERVAN 2005]JERVAN, G. *Hybrid Built-in Self-test and Test Generation Techniques for Digital Systems*. [S.l.]: Dept. of Computer and Information Science, Univ., 2005.

[JERVAN G. ; MARKUS]JERVAN G. ; MARKUS, A. . P. P. . R. J. . U. R. Turbo Tester: A CAD System for Teaching Digital Test. *Microelectronics Education*, p. 287–290.

[JUTMAN]JUTMAN, A. On SSBDD Model Size & Complexity. In: *Proc. of 4th Electronic Circuits and Systems Conference (ECS'03)*. [S.l.: s.n.]. p. 11–12.

[JUTMAN A. ; RAIK]JUTMAN A. ; RAIK, J. . U. R. SSBDDs: Advantageous Model and Efficient Algorithms for Digital Circuit Modeling, Simulation & Test. In: *Proc. of 5th International Workshop on Boolean Problems (IWSBP'02)*. [S.l.: s.n.]. p. 19–20.

[KOCAN e SAAB 2001]KOCAN, F.; SAAB, D. ATPG for combinational circuits on configurable hardware. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 9, n. 1, p. 117–129, 2001.

[KOCAN e SAAB 2007]KOCAN, F.; SAAB, D. Dynamic Fault Diagnosis of Combinational and Sequential Circuits on Reconfigurable Hardware. *Journal of Electronic Testing*, Springer, v. 23, n. 5, p. 405–420, 2007.

[KRSTIC A.; CHENG 1998]KRSTIC A.; CHENG, K. T. *Delay Fault Testing for VLSI Circuits*. [S.l.]: Kluwer Academic Publishers, 1998.

[LALA 1997]LALA, P. *Digital Circuit Testing and Testability*. [S.l.]: Academic Pr, 1997.

[LOOSEMORE S. ; STALLMAN 1993]LOOSEMORE S. ; STALLMAN, R. . M. R. . O. A. . D. U. *The GNU C Library Reference Manual*. [S.l.]: Free Software Foundation, 1993.

[LUBASZEWSKI M.; COTA 2002]LUBASZEWSKI M.; COTA, E. K. M. Teste e Projeto Visando o Teste de Circuitos e Sistemas Integrados. *REIS, RA da L.(Ed.) Concepção de Circuitos Integrados*, v. 2, p. 167–189, 2002.

[MARTINEZ 2007]MARTINEZ, D. A. *Speeding-up model-based fault injection of deep-submicron CMOS faults models through dynamic and partially reconfigure FPGAs*. [S.l.]: Universidad Politécnica de Valencia, 2007.

[MORAES 2006]MORAES, M. *STEP: planejamento, geração e seleção de auto-teste on-line para processadores embarcados*. [S.l.]: UFRGS, 2006.

[MOURAD S. ; ZORIAN 2000]MOURAD S. ; ZORIAN, Y. *Principles of Testing Electronic Systems*. [S.l.]: Wiley-Interscience, 2000.

[PELLEGRINI et al. 2008]PELLEGRINI, A. et al. CrashTest: A Fast High-Fidelity FPGA-Based Resiliency Analysis Framework. In: *IEEE International Conference on Computer Design, September*. [S.l.: s.n.], 2008.

[RAIK 2001]RAIK, J. *Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams*. Tese (Doutorado) — Tallinn Technical University, 2001.

[ROTH 1966]ROTH, J. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, v. 10, n. 4, p. 278–291, 1966.

[RUPPERT G.C.S. ;GEUS 2004]RUPPERT G.C.S. ;GEUS, P. Uma Análise de Segurança das Versões do Protocolo NFS. 2004.

[SABADE S.S. ; WALKER 2004]SABADE S.S. ; WALKER, D. I DDX-based test methods: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, ACM New York, NY, USA, v. 9, n. 2, p. 159–198, 2004.

[SLOSS A.N.; SYMES 2004]SLOSS A.N.; SYMES, D. . W. C. *ARM System Developer's Guide: Designing and Optimizing System Software*. [S.l.]: Morgan Kaufmann, 2004.

[SOUZA 2006]SOUZA, D. *Microcontroladores ARM7 (Philips - família LPC213x) - O poder dos 32 Bits - Teoria e Prática*. [S.l.]: Érica, 2006.

[SWANSON B. ; GOSWAMI 2006]SWANSON B. ; GOSWAMI, D. *Using Timing Constraints For Generating At-Speed Test Patterns*. 2006.

[TECHNOLOGIC]TECHNOLOGIC. "ts-7300 manual hardware & software". disponível por www em:$< http : //www.embeddedarm.com/documentation/ts-7300-manual.pdf >$. acesso em 5 de maio, 2008.

[TINY-TECH 2006]TINY-TECH. *TB500A Datasheet*. [S.l.], 2006.

[WANG L.T. ; WU 2006]WANG L.T. ; WU, C. . W. X. *VLSI Test Principles and Architectures: Design for Testability (Systems on Silicon)*. [S.l.]: Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006.

[WHITLEY 1994]WHITLEY, D. A genetic algorithm tutorial. *Statistics and Computing*, Springer, v. 4, n. 2, p. 65–85, 1994.

[XILINX 2004]XILINX. *Spartan-3 FPGA Family: Introduction and Ordering Information*. [S.l.], 2004.

ANEXO A - Análise temporal em software dos *benchmark* ISCAS85

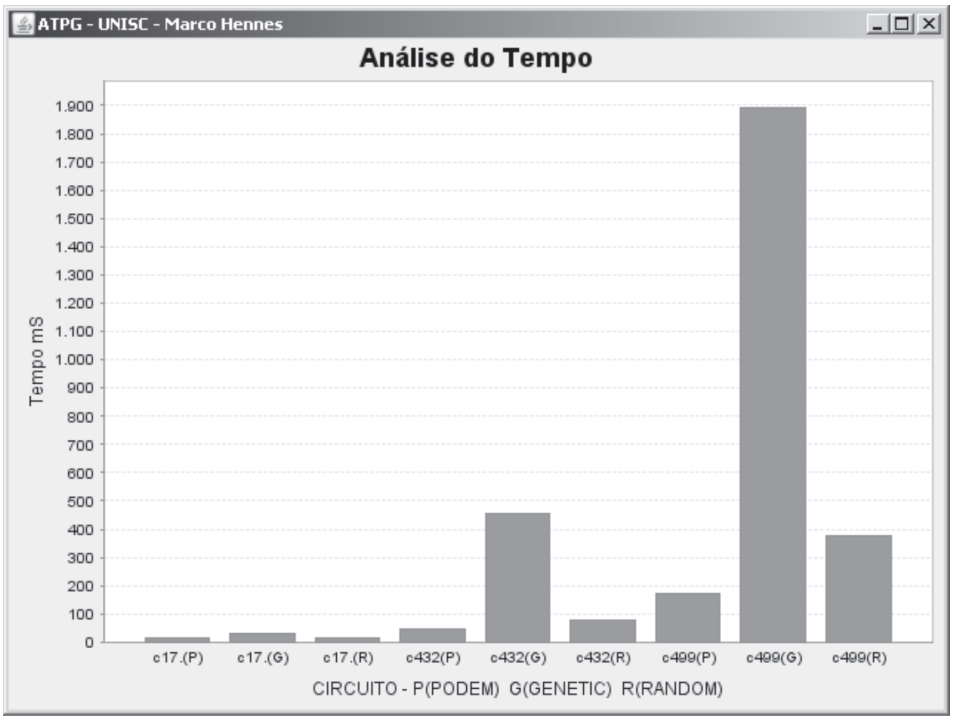


Figura 48: Análise temporal conforme o algoritmo dos circuitos c17, c432 e c499

Na Figura 48 são mostrados os circuitos c17, c432 e c499 usando-se os algoritmos PODEM(P), GENETIC(G) e RANDOM(R). Como o circuito c17 é um circuito pequeno, apresenta uma pequena diferença de tempo computacional para os algoritmos. Já os circuitos c432 e c499 apresentam um tempo computacional muito maior para o algoritmo GENETIC se comparado com os outros dois. Porém, o algoritmo GENETIC consegue um quantidade menor de vetores, se comparado ao PODEM e RANDOM. No circuito c499 o algoritmo GENETIC gera 86 vetores de testes, enquanto que o PODEM gera 123, com a mesma taxa de cobertura, conforme mostrado nas Tabelas 10 e 11 das páginas 72 e 73.

Analisando-se a Figura 49 é possível constatar como anteriormente que o algoritmo GENETIC possuí um tempo muito superior se comparado aos outros algoritmos. Contudo o circuito c1355 apresenta quase um mesmo tempo computacional para os algoritmos devido a simplicidade do teste. Já o algoritmo RANDOM consegue um tempo maior que o PODEM e na maioria dos casos um quantidade de vetores de teste maior com um taxa de cobertura semelhante.

A Figura 50 mostra os circuitos c2670, c3540 e o c5315. Os circuitos c3540 e c5315 são circuitos maiores e possuem mais de 1200 *gates*. Contudo, o algoritmo PODEM consegue

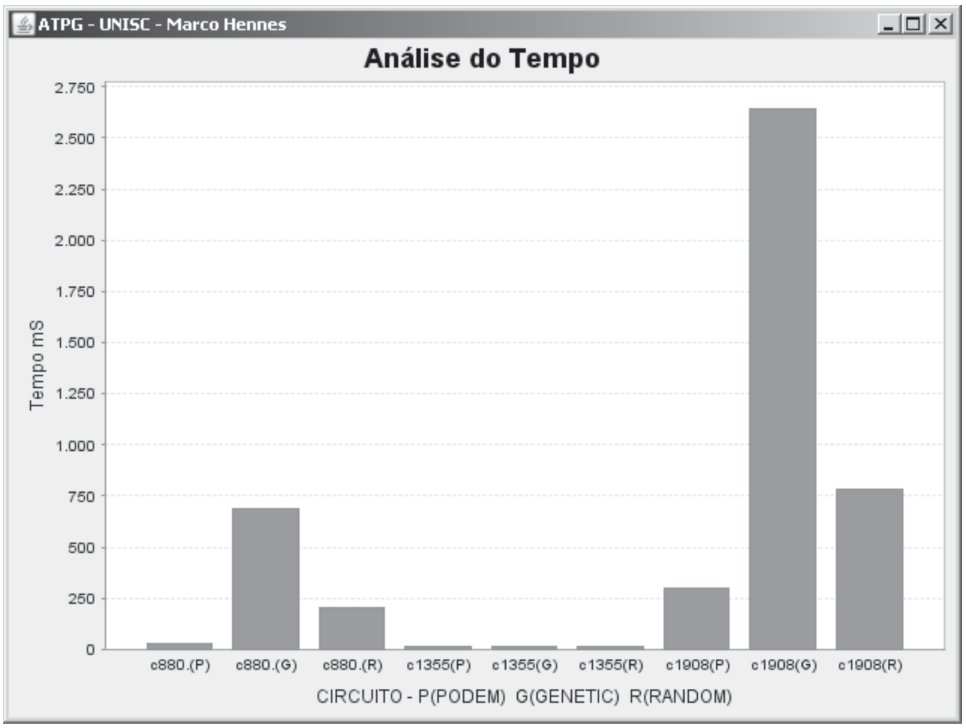


Figura 49: Análise temporal conforme o algoritmo dos circuitos c880, c1355 e c1908

um tempo computacional muito inferior se comparado ao GENETIC. Isso acontece pelo fato do método de busca do algoritmo PODEM estar baseado na quantidade de portas de entrada e não no número de *gates*.

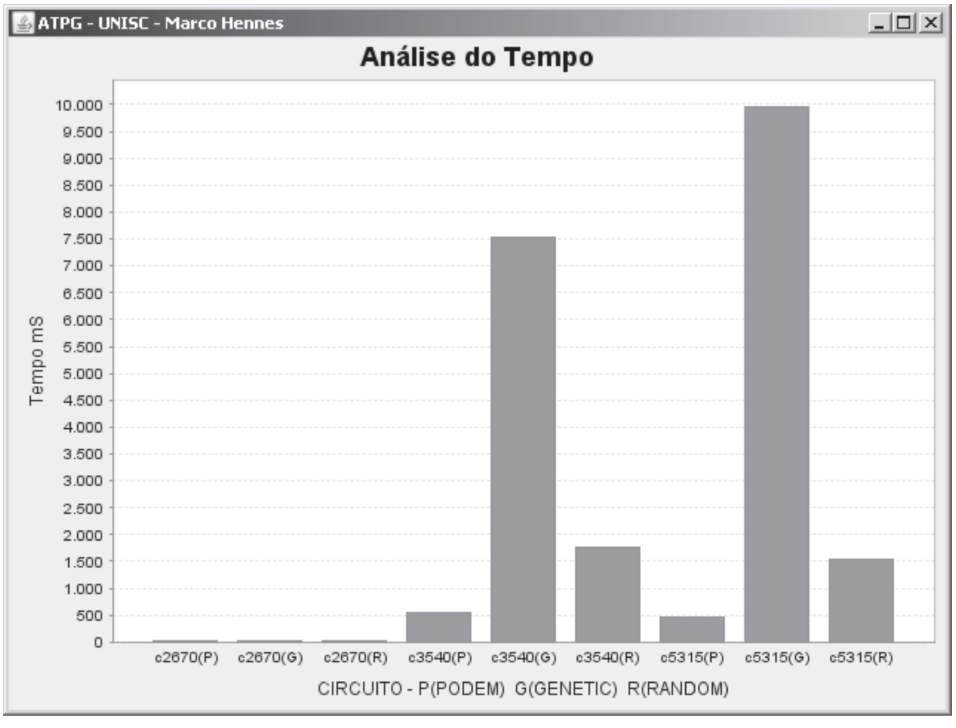


Figura 50: Análise temporal conforme o algoritmo dos circuitos c2670, c3540 e c5315

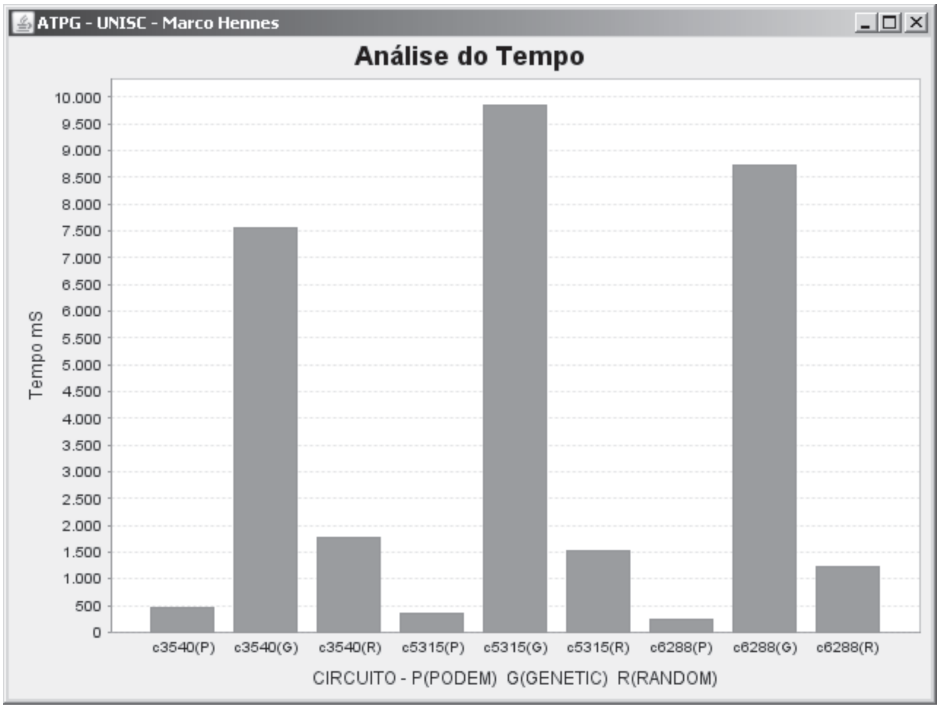


Figura 51: Análise temporal conforme o algoritmo dos circuitos c3540, c5315 e c6288

ANEXO B - Análise da taxa de cobertura dos *benchmark* ISCAS85

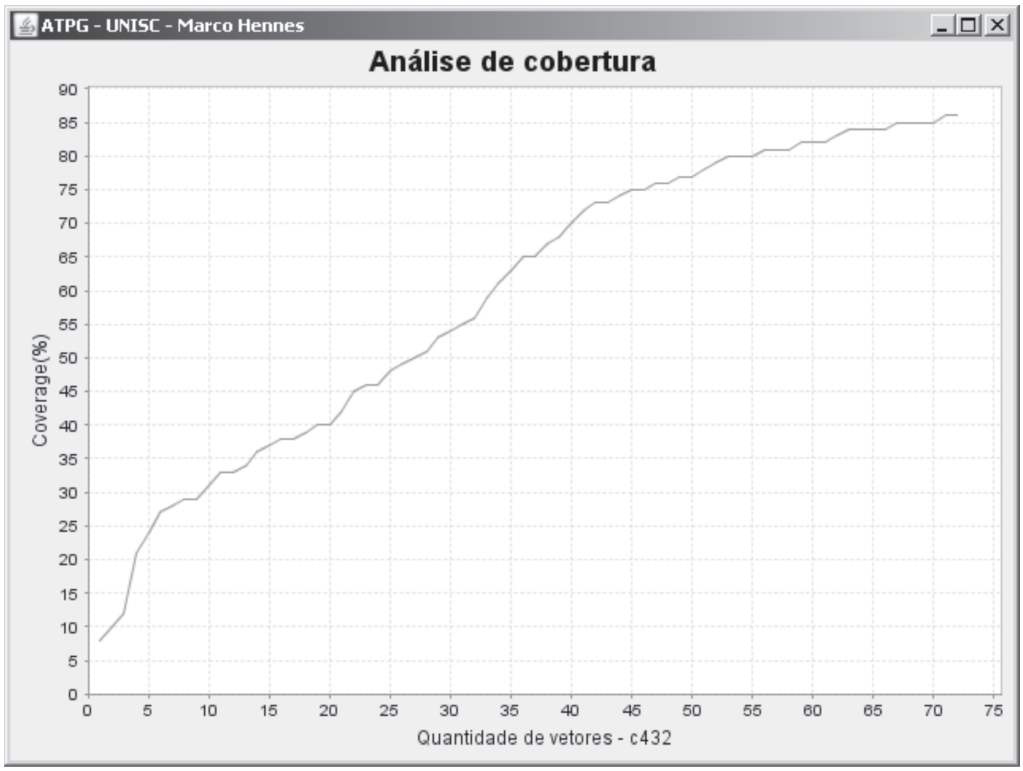


Figura 52: Análise de cobertura utilizando o algoritmo PODEM do circuito c432

Nas Figuras 52, 53, 54, 55, 56 e 57 é mostrado às análises da taxa de cobertura dos circuitos conforme a quantidade de vetores de testes. Foi realizado somente testes para o algoritmo PODEM, devido a grande quantidade de gráficos gerados para os 3 algortimos. Na Figura 52 é mostrado o circuito c432 e a sua taxa de cobertura é proporcional a quantidade de vetores, ou seja, é bem próxima de uma reta. Já na Figura 53 é possível ver o circuito c880 e com uma taxa de cobertura próxima de 90% com uma quantidade de 35 vetores. Com aproximadamente 80 vetores se consegue uma taxa de 100%. Dessa forma, com mais que o dobro de vetores, a taxa de cobertura aumentou somente 10%.

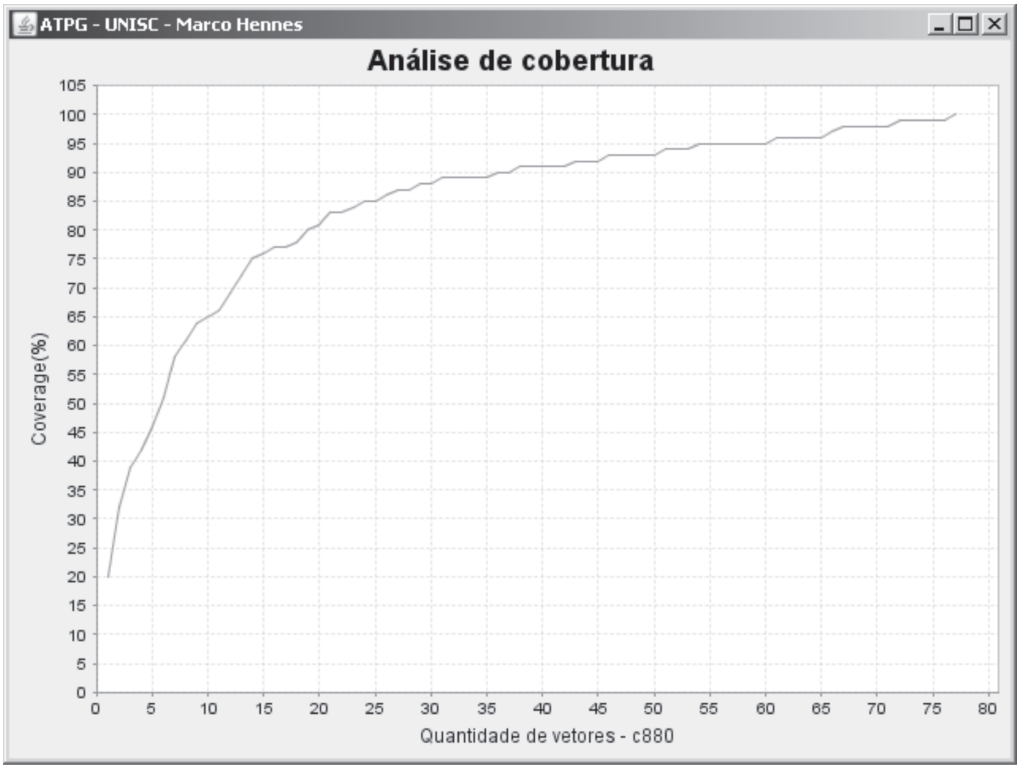


Figura 53: Análise de cobertura utilizando o algoritmo PODEM do circuito c880

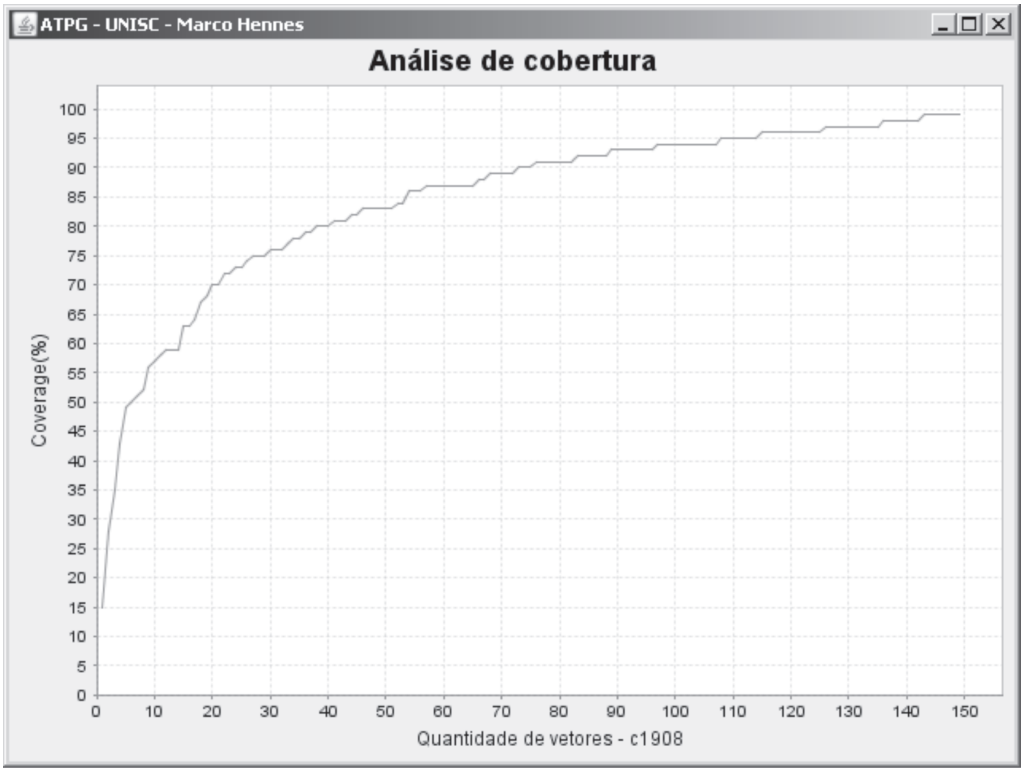


Figura 54: Análise de cobertura utilizando o algoritmo PODEM do circuito c1908

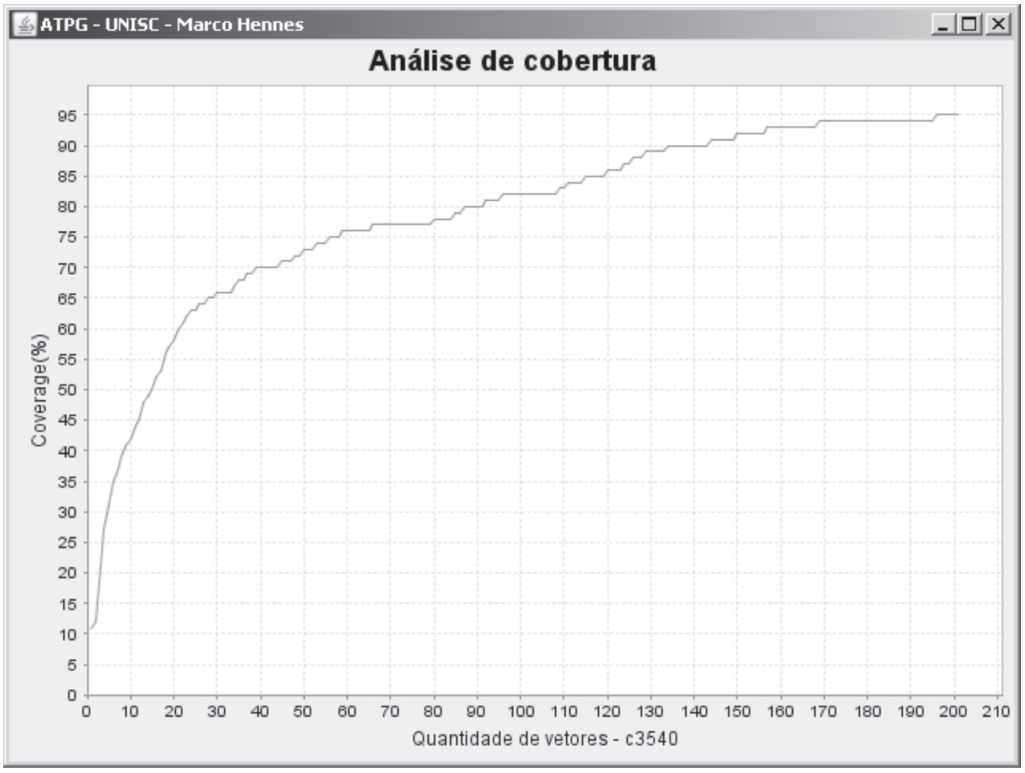


Figura 55: Análise de cobertura utilizando o algoritmo PODEM do circuito c3540

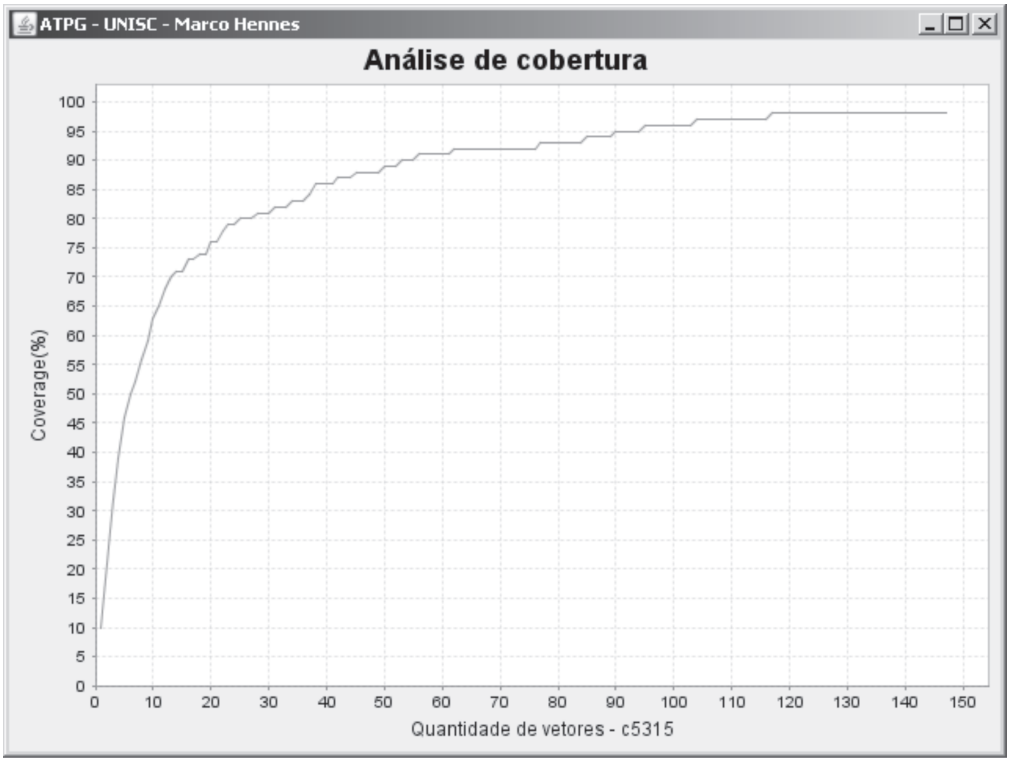


Figura 56: Análise de cobertura utilizando o algoritmo PODEM do circuito c5315

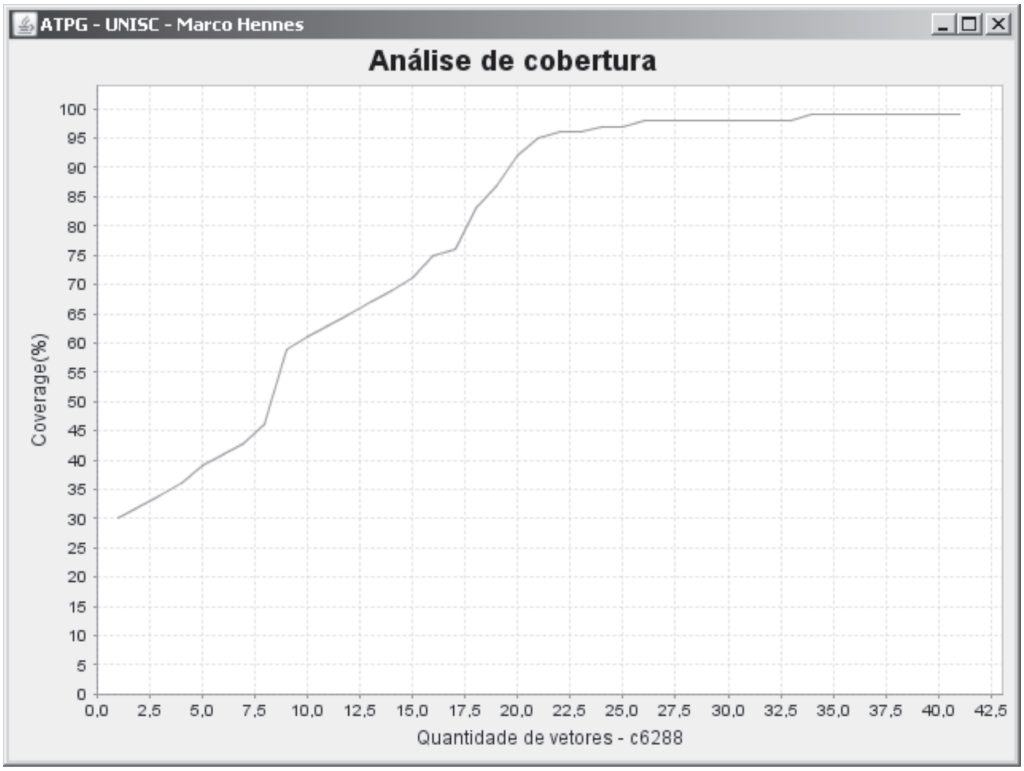


Figura 57: Análise de cobertura utilizando o algoritmo PODEM do circuito c6288

ANEXO C - Circuitos do *benchmark* ISCAS85

Neste anexo é descrito os circuitos do *benchmark* ISCAS85 usados em todo o trabalho. A Tabela 13 mostra os circuitos e a quantidade de portas lógicas em cada um.

Tabela 13: Descrição dos circuitos ISCAS85

Circuito	Descrição do circuito	buffer	not	and	nand	or	nor	xor	Total
c17	Portas Lógicas				6			18	6
c432	Corretor(32 bits)		40	4	79	2	19	104	160
c499	Corretor(32 bits)		40	56		29			202
c880	ALU de 8 bits	26	63	117	87	2	61		383
c1355	Corretor(32 bits)	32	40	56	416				546
c1908	Detector e corretor(16 bits)	162	277	63	377	92	1		880
c2670	ALU de 12 bits	196	321	333	254	214	12		1193
c3540	ALU de 8 bits	223	490	498	298		68		1669
c5315	ALU de 9 bits	313	581	718	454	244	27		2307
c6288	ALU de 9 bits		32	256			2128		2416

A informação dos circuitos foi retirado a partir de http://www.eecs.umich.edu/∼jhayes/iscas/ onde uma mais recente publicação sobre essa engenharia reversa é mostrada em (HANSEN et al. 1999).

ANEXO D - IBERCHIP XV Workshop /Bs As, Argentina March 25-27,2009

Plataforma para testes de circuitos digitais

Marco A. B. Hennes, Rafael R. dos Santos, Rolf F. Molz
Universidade de Santa Cruz do Sul - UNISC
96815-900 - Santa Cruz do Sul - RS - Brasil
Email:hennes@mx2.unisc.br, {rsantos, rolf}@unisc.br

***Resumo*—Esse artigo descreve uma plataforma de testes de circuitos integrados. Essa análise é feita em circuitos combinacionais do *benchmark* ISCAS85 para falhas do tipo *stuck-at*. A plataforma de testes foi desenvolvida com componentes de baixo custo, o que dá à plataforma um grande diferencial. Além disso, o ambiente realiza tanto testes em software, como em hardware. Para os testes realizados em hardware, o ambiente emula o circuito em um FPGA com o propósito de se comprovar os testes em simulação em um ambiente real. Outro diferencial importante, é a integração dos ambientes de simulação, permitindo realizar testes em software, como em hardware.**

I. Introdução

Com o crescimento da complexidade dos circuitos VLSI(*Very Large Scale Integration*) e SOC(*System-on-Chip*), a geração de testes acabou se tornando uma das etapas mais complicadas e demoradas no domínio da concepção de circuitos integrados. Quanto mais complexos se tornam os sistemas eletrônicos, mais importante se torna as etapas de verificação e teste. Um circuito integrado durante seu processo de produção é submetido a diferentes tipos de testes. Um dos testes mais utilizados é o teste estrutural, baseado em falhas [1], o qual permite a uma medida quantitativa da cobertura de falhas do circuito.

A plataforma proposta foi usada para testes de circuitos digitais. A plataforma proposta foi implementada em Java e usa mais duas ferramentas EDA(*Electronic Design Automation*) para se testar os circuitos. É possível dividir essa plataforma em testes em software e em hardware. Nos testes em software é possível, com uma descrição de um circuito, usar algoritmos para se testar esse circuito. Já nos testes em hardware, foi desenvolvida uma plataforma de baixo custo que usa um microprocessador ARM e um FPGA. Tanto os testes em software, como os em hardware, tem o propósito de se conseguir um circuito que funcione corretamente. Nos dois casos são aplicados vetores de teste na entrada do circuito e são analisadas suas saídas. Um dos grandes desafios está em se conseguir um conjunto reduzido de vetores de teste que consigam uma grande cobertura do circuito. Nos testes em hardware, o ambiente emula o circuito em um FPGA com o propósito de se comprovar os testes em simulação em um ambiente real. Este trabalho está organizado do seguinte modo: a Seção II apresenta trabalhos relacionados, enquanto que a Seção III apresenta algumas definições importantes na área de testes de sistemas digitais e introduz a metodologia usada. A Seção IV, mostra os tipos de algoritmos usados em testes e os modelos de representação. A Seção V fala sobre a arquitetura de testes proposta e a seção VI mostra os resultados obtidos. Finalmente a Seção VII discute os resultados alcançados na verificação de circuitos digitais.

II. Trabalhos correlatos

Nessa seção se descreve alguns trabalhos relacionados a testes de circuitos digitais. A utilização de FPGA para se testar circuitos é mostrada por Kocan [2]. Nesse trabalho é mostrado quantas vezes se consegue aumentar a velocidade, ou seja, *speeds up*, usando a emulação em hardware comparado com a análise em software. Kocan mostrou um método para se emular circuitos combinacionais e sequênciais para falhas *stuck-at*. Esse método demonstra a análise de falhas usando hardware. Ellervee [3] mostrou também um aumento na velocidade de emulação se comparado com a análise de falhas em software. Porém, Ellervee mostra análises somente para circuitos combinacionais.

Schneider [4] mostrou um ambiente que integra várias ferramentas para testes de circuitos usando Java.

III. Conceitos Fundamentais

Para se testar os circuitos foi utilizado o teste estrutural, que depende de uma estrutura específica (portas lógicas, netlist) do circuito. Umas das grandes vantagens do teste estrutural é que ele nos permite desenvolver algoritmos. Esses algoritmos são baseados no modelo de falhas [5]. Uma ferramenta EDA foi usada para a geração automática dos vetores de testes. Na área de testes de circuitos integrados a geração automática de vetores é conhecida como ATPG(*Automatic Test Pattern Generation)*. A geração do teste estrutural só pode acontecer se o modelo em níveis de portas lógicas for disponibilizado [6]. No caso de se ter essa informação, uma alta taxa de cobertura de falhas é possível com uma pequena quantidade de vetores. A modelagem de falhas está intimamente ligada à modelagem do circuito. O modelo de falhas *stuck-at* é o modelo mais usado em testes de circuitos digitais [5].

Falhas lógicas são modeladas como falhas *stuck-at*. Uma falha *stuck-at* força um valor fixado (0 ou 1) para uma linha de sinal de um circuito, onde essa linha pode ser a entrada ou saída de uma porta lógica ou *flip-flop* [7]. Esse *stuck-at* pode ser um *stuck-at-1*(s-a-1) ou um *stuck-at-0*(s-a-0). Esse modelo é simples e independe da tecnologia usada. Um circuito pode ter várias falhas *stuck-at*. Uma linha de sinal de um circuito pode ter um s-a-0, s-a-1 ou livre de falhas(*fault-free*). Um circuito com n linhas de sinal pode ter 3^{n-1} possibilidades de linhas *stuck-at*. Então, para um circuito de tamanho moderado já é possível ter um grande número de falhas *stuck-at*. Por isso

Figura 58: IBERCHIP XV Workshop (1/5)

é uma prática comum considerar somente o modelo de falhas *single stuck-at* . Esse modelo assume que somente uma entrada ou saída de uma porta lógica ou *flip-flop* pode estar s-a-0 ou s-a-1. O número máximo de falhas *single stuck-at* para um circuito com n linhas é de $2 * n$ [5].

O número de falhas, usando o modelo de falhas *single stuck-at* é consideravelmente reduzido. Analisando-se a Figura 1, observa-se uma porta lógica NAND com uma entrada A com s-a-1. Essa entrada sempre permancerá em nível lógico 1, resultando em um curto-circuito para o nível lógico 1. A saída dessa NAND é 0 quando a entrada B estiver em 1 e quando o s-a-1 em A estiver presente. Já se esse circuito não possuir o s-a-1 ou s-a-0, ou seja, for *fault-free*, a saída desse circuito será 1 para as entradas mostradas na Figura 1. Então, se for usado na entrada desse circuito A=0 e B=1, pode-se testar o s-a-1 em A, porque existe uma diferença nas saídas do circuito *fault-free* e o *faulty gate* [8].

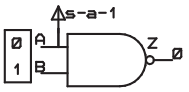


Figura 1. Porta lógica NAND

Devido o que foi descrito acima(modelo simples e independente da tecnologia), foi escolhido para testes o modelo de falhas *single struck-at*. Dessa forma toda a metodologia usada neste trabalho está baseada no modelo de falhas *single struck-at*.

Para um circuito existem $2 * n$ falhas *single stuck-at*, porém muitas dessas falhas são equivalentes. Se for analisado um circuito como se fossem simples portas lógicas, então é possível analisar grandes circuitos. Considerando uma porta lógica AND conforme visto na Figura 2a, qualquer falha do tipo s-a-0 na saída ou entrada dessa porta sempre terá uma saída constante em 0. Dessa forma é possível dizer que essas falhas s-a-0 são equivalentes. Uma análise similar para outras portas lógicas é possível ver na Figura 2b, 2c, 2d e 2e [5].

IV. Geração de testes para circuitos digitais

Para se analisar um circuito digital é necessário um modelo matemático para representá-lo. O modelo matemático usado nesse trabalho é o SSBDD devido a ferramenta Turbo Tester [9] usá-lo e ser um modelo bem aceito, diferentemente de outros modelos BDD [10] [11]. Com o modelo matemático escolhido para representar o circuito, ainda é necessário o modelo do tipo de falhas a ser testado. O modelo de falhas usado foi o modelo de falhas *single stuck-at*. Com o modelo matemático escolhido e o modelo da simulação de falhas, a geração automática de vetores é o caminho ideal para se conseguir um procedimento de teste eficiente para um determinado circuito. A geração automática de testes tenta conseguir a máxima cobertura de falhas usando vetores de testes nas entradas e analisando as saídas do circuito. A

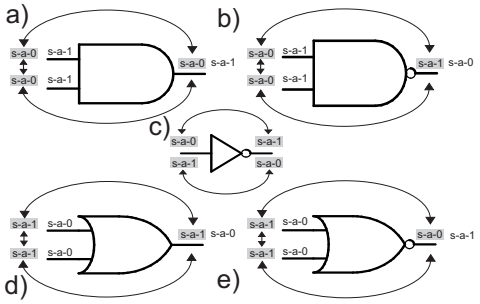


Figura 2. Equivalência de falhas

geração desses vetores de testes, no domínio digital, foi feita através de algoritmos. O algoritmo D proposto por Roth [12], precursor na área de testes, é o fundamento do algoritmo PODEM [13] usado nesse trabalho. Além desse algoritmo, também foram usados os algoritmos GENETIC [14][5] e RANDOM [15].

A simulação de falhas (*fault simulation*) consiste no modelamento do circuito na presença de falhas. Comparando a resposta do circuito com falhas e do circuito sem falhas, usando o mesmo vetor de teste, pode-se determinar as falhas detectadas por esse vetor [16]. A simulação de falhas classifica as falhas em um circuito como **detectáveis** ou **indetectáveis**. A lista de todas as falhas é feita com o modelo *single stuck-at* retirando as falhas equivalentes conforme mostrado na Figura 2. A simulação é inicializada com os vetores de verificação fornecidos e produz uma lista de falhas. A cobertura dessas falhas, ou seja, a cobertura de falhas é mensurada através do número de falhas detectadas e do total de falhas que consta na lista de falhas, conforme mostrado na equação 1.

$$Cobertura\,de\,falhas = \frac{Quantidade\,de\,f.\,detectaveis}{Total\,de\,falhas} \quad (1)$$

Quando se consegue uma alta cobertura de falhas, a simulação pode terminar. Se existirem vetores que não podem ser simulados, estes podem ser removidos para diminuir a lista de vetores de teste. Porém, se esses vetores de teste não produzirem uma cobertura adequada conforme a equação 1, algumas dessas falhas não detectadas da lista de falhas podem ser passadas a um programa de testes para se produzir novos vetores [5]. Um circuito C() é livre de falhas e um circuito C(f_1) até um C(f_n) são cópias com falhas inseridas. O mesmo vetor é aplicado para todos os circuitos e as saídas dos circuitos com falhas são comparadas com o circuito livre de falhas, conforme mostrado na Figura 3.

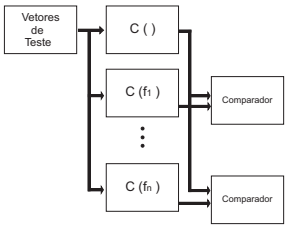


Figura 3. Simulação de falhas

V. Arquitetura Proposta (ARM e FPGA)

Um dos problemas de se testar um circuito integrado em um ATE[1](*automatic test equipment*) está em seu alto custo para aquisição. Para se resolver isso, foi desenvolvida uma plataforma de testes de baixo custo, conforme mostrado na Figura 4, onde essa pode ser dividida em duas:testes em software e em hardware. Foi desenvolvida uma plataforma gerenciadora, na linguagem Java, de ambos os tipos de testes. Na parte de software foi usado o programa Leonardo Spectrum e o Turbo Tester em conjunto com o Java. A versão usada do Turbo Tester possui diferentes algoritmos para se testar e analisar os circuitos, porém não possui um ambiente GUI(*Graphical User Interface*). A ferramenta desenvolvida em Java cobre essa lacuna. Já em hardware foi usado um microcontrolador da família ARM e um FPGA para emular o hardware. O ARM foi escolhido devido a sua alta capacidade de processamento.

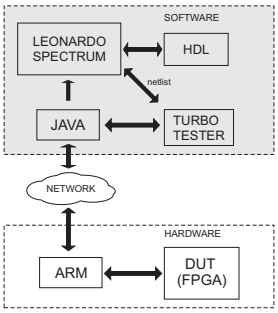


Figura 4. Arquitetura Proposta

Inicialmente, para se testar um circuito é preciso tê-lo descrito em linguagem HDL(*Hardware Description Language*). Foram usado para testes os circuitos do benchmark ISCAS85. A partir da aplicação em Java (Figura 5), converte-se a descrição HDL do circuito a ser testado em um *netlist*. Para isso a plataforma em Java acessa o programa Leonardo Spectrum e faz a síntese do circuito para ASIC e o transforma em um *netlist* do circuito. Essa descrição em *netlist* do circuito é chamada de arquivo EDIF. Para esse EDIF foi desenvolvida uma biblioteca que contém a descrição das portas lógicas primitivas do circuito. Com a ferramenta de Java é acessado o programa Turbo Tester e mais a biblioteca das primitivas e se faz a conversão do arquivo EDIF num arquivo SSBDD. Esse arquivo é uma representação da lógica do circuito digital que será usado para a simulação do circuito. Com esses arquivos convertidos em SSBDD é possível usar as ferramentas do Turbo Tester para fazer as simulações .

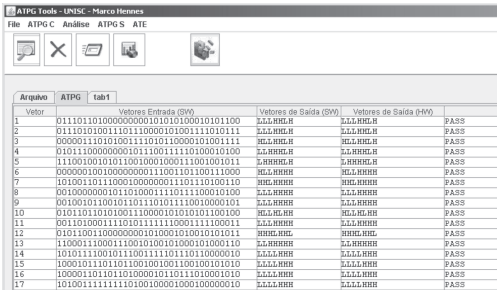


Figura 5. Programa desenvolvido em Java

Para acelerar essa simulação e também fazer uma análise real em hardware, é possível usar um FPGA para emular o DUT conforme mostrado na Figura 4. A ferramenta em Java envia por *socket*[2] os vetores de testes para o microcontrolador ARM. Na figura 6 é mostrada a comunicação usando *socket*. O *socket* foi implementado em Java no microcomputador(server) e em linguagem C no ARM(client).

O ARM recebe esses dados, e acaba serializando esses dados e injetando no FPGA somente por um pino de I/O. Essa técnica de se serializar os dados foi usada, devido a certos circuitos de testes possuírem muitos mais entradas do que os pinos de I/O do ARM. No FPGA esses dados são recebidos e desserializados por um *shift register* conforme mostrado na Figura 7(a). Esse *shift register* é controlado por um clock do ARM. Após isso, os dados são injetados no circuito e são recebidos por um outro *shift regiter* que serializa os dados novamente conforme mostrado na Figura 7(b). O ARM recebe esses dados e envia esses resultados dos testes dos circuitos por

[1]O ATE é um aparelho usado para aplicar vetores de testes a um DUT (device-under-test) e analisado as respostas do DUT e marcado esse DUT como defeituoso ou não defeituoso.

[2]*Sockets* é um canal generalizado de comunicação entre processos e suporta comunicação entre processos independentes, e mesmo entre processos rodando em diferentes máquinas se comunicando na rede.

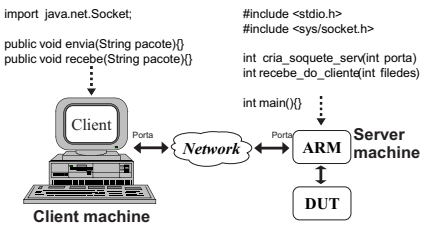


Figura 6. Comunicação entre ARM e microccomputador usando *socket*

socket para a ferramenta em Java, que compara esses vetores com vetores *fault free* e classifica o circuito como bom ou ruim. Na Figura 7 o DUT que está sendo testado é o circuito c17 do *benchmark* ISCAS85.

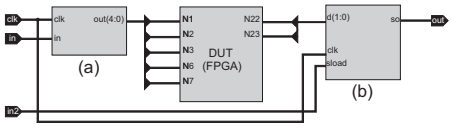


Figura 7. Arquitetura do DUT

A ferramenta de testes desenvolvida possui um ambiente gráfico para análise dos circuitos. Existem dois tipos de gráfico para análise. O primeiro, que é o gráfico em barras, pode ser usado para a medição de tempo da geração do vetores de teste. É possível comparar circuitos diferentes e também é possível usar algoritmos diferentes conforme mostrado na Figura 8. Nessa figura, é mostrado o teste de 3 circuitos diferentes,

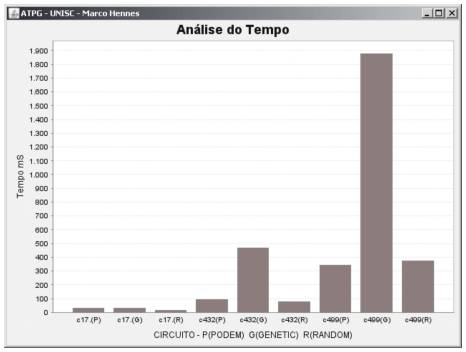


Figura 8. Análise de tempo

onde para cada circuito foi aplicado um algoritmo diferente (PODEM, GENETIC, RANDOM). Também é possível uma análise da cobertura de um circuito conforme a quantidade de vetores testados, conforme é mostrado na Figura 9. Nessa figura, apresenta-se uma análise de cobertura do circuito c432 conforme a quantidade de vetores.

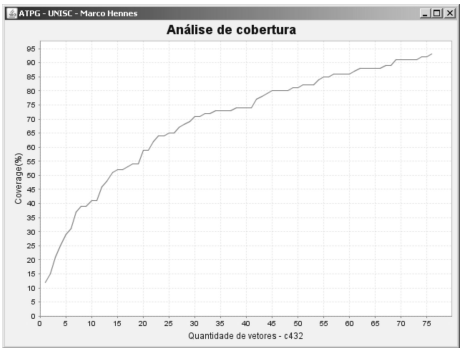


Figura 9. Análise de cobertura

VI. Resultados Experimentais

Nessa seção é investigado o tempo requerido para se testar os circuitos em software e hardware. Foram usados para os testes os *benchmarks* ISCAS85. O programa ISE 9.1 (Xilinx) foi usado para a síntese dos circuitos para a FPGA. O circuito sintetizado é implementado numa Spartan 3 [17] da Xilinx de 50 MHz de *clock*. O ARM usado no hardware para injetar os sinais na FPGA é o EP9302 [18] da Cirrus Logic de 200 MHz de *clock*.

Na Tabela I é mostrado a análise em software para os circuitos do *benchmark* ISCAS85. A simulação foi realizada num Pentium 4 de 3GHz e 512MB de RAM. Nessa Tabela, apresenta-se os circuitos e seus respectivos tempos(T) para simulação e a taxa de cobertura(TC), conforme o algoritmo usado. Esse tempo leva em conta o tempo da plataforma desenvolvida em Java para se acessar o Turbo Tester, mais o tempo da ferramenta Turbo Tester para se analisar o circuito. Dessa forma, a cobertura será a mesma alcançada pela ferramenta Turbo Tester, porém o tempo será a soma dos tempos das duas ferramentas.

Na tabela II são mostrados os resultados iniciais para a emulação em hardware. Os vetores que são usados nos testes em hardware, são os mesmos usados em software e são transferidos por *socket* pela ferramenta em Java. Dessa forma, a parte dos testes em hardware, não possui o gerador de vetores de testes e a análise das saídas dos circuitos emulados é feita no Java. É possível notar que existe um menor tempo na Tabela II, se comparado os mesmos circuitos e algortimos mostrados na Tabela I. Isso se deve ao fato dos testes em hardware,

serem mais rápidos do que os em software, apesar de a parte em hardware não possuir o gerador de vetores. Porém, deve-se salientar que esse tempo visto na Tabela II leva em conta o tempo de se enviar e receber dados por *socket*, o do ARM tratar os dados na entrada e saída do DUT e o de propagação no DUT.

Tabela I
TAXA DE COBERTURA E TEMPO DE SIMULAÇÃO

Circuito	PODEM		GENETIC		RANDOM	
	T(ms)	TC	T(ms)	TC	T(ms)	TC
c17	15	100%	21	100%	16	100%
c432	47	86,20%	453	93,01%	78	93,01%
c499	172	99,63%	1860	99,63%	375	99,63%
c880	78	100%	1265	100%	329	99,39%
c1355	16	99,63%	16	99,63%	15	99,63%
c1908	234	99,53%	2579	99,60%	750	99,56%
c2670	16	95,01%	16	95,01%	16	95,01%
c3540	516	95,33%	7390	95,68%	1718	95,59%
c5315	750	98,98%	9703	99,29%	1516	99,29%
c6288	235	99,30%	8524	99,30%	1187	99,30%

Tabela II
TEMPO DE EMULAÇÃO EM HARDWARE

Circuito	PODEM Tempo(ms)	GENETIC Tempo(ms)	RANDOM Tempo(ms)
c17	>0	>0	>0
c432	31	31	31
c499	78	109	109
c880	78	63	79

VII. CONCLUSÕES

Esse artigo apresentou uma nova plataforma de testes para circuitos combinacionais. Um grande diferencial dessa plataforma, é que ela utiliza somente componentes de baixo custo. Outro grande diferencial é a integração de várias ferramentas, para testes em software e em hardware.
Os resultados mostrados na Tabela II, ainda estão em desenvolvimento. Como trabalho futuro pretende-se implementar essa plataforma para testes de circuitos seqüências.

VIII. AGRADECIMENTOS

Os autores agradecem à Universidade de Santa Cruz do Sul (UNISC) e a CAPES pelo apoio para a realização desse trabalho. Esse trabalho foi realizado nas dependências da UNISC, dentro do laboratório do GPSEM(Grupo de Projetos e Sistemas Embarcados).

REFERÊNCIAS

[1] M. Lubaszewski, E. Cota, and M. Krug, "Teste e Projeto Visando o Teste de Circuitos e Sistemas Integrados," *REIS, RA da L.(Ed.) Concepção de Circuitos Integrados*, vol. 2, pp. 167–189, 2002.
[2] F. Kocan and D. Saab, "Dynamic Fault Diagnosis of Combinational and Sequential Circuits on Reconfigurable Hardware," *Journal of Electronic Testing*, vol. 23, no. 5, pp. 405–420, 2007.
[3] P. Ellervee, J. Raik, and V. Tihhomirov, "Environment for Fault Simulation Acceleration on FPGA."
[4] A. Schneider *et al.*, "Internet-based Collaborative Test Generation with MOSCITO," in *Proc. of DATE02*, pp. 4–8.
[5] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-signal VLSI Circuits*. Kluwer Academic, 2000.
[6] D. Gizopoulos and Y. Paschalis, A.; Zorian, *Embedded Processor-Based Self-Test*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2004.
[7] I. Berger and Z. Kohavi, "Fault Detection in Fanout-Free Combinational Networks," *Transactions on Computers*, vol. 100, no. 22, pp. 908–914, 1973.
[8] P. Lala, *Digital Circuit Testing and Testability*. Academic Pr, 1997.
[9] G. Jervan, A. Markus, P. Paomets, J. Raik, and R. Ubar, "Turbo Tester: A CAD System for Teaching Digital Test," *Microelectronics Education*, pp. 287–290.
[10] A. Jutman, J. Raik, and R. Ubar, "SSBDDs: Advantageous Model and Efficient Algorithms for Digital Circuit Modeling, Simulation & Test," in *Proc. of 5th International Workshop on Boolean Problems (IWSBP'02)*, pp. 19–20.
[11] A. Jutman, "On SSBDD Model Size & Complexity," in *Proc. of 4th Electronic Circuits and Systems Conference (ECS'03)*, pp. 11–12.
[12] J. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278–291, 1966.
[13] P. Goel, "An implicit enumeration algorithm to generate tests for combinational circuits," *IEEE Transactions on Computers*, vol. 30, no. 3, pp. 215–222, 1981.
[14] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, vol. 4, no. 2, pp. 65–85, 1994.
[15] S. Mourad and Y. Zorian, *Principles of Testing Electronic Systems*. Wiley-Interscience, 2000.
[16] M. Cilleti, *Advanced digital design with the Verilog HDL*. Prentice Hall, 2002.
[17] *Spartan-3 FPGA Family: Introduction and Ordering Information*, ds099-1.pdf, Xilinx, 2004.
[18] *EP9302 Data Sheet*, EP9302_PP3.pdf, Cirrus Logic, 2005.

Figura 62: IBERCHIP XV Workshop (5/5)

ANEXO E - Placa de circuito impresso projetada

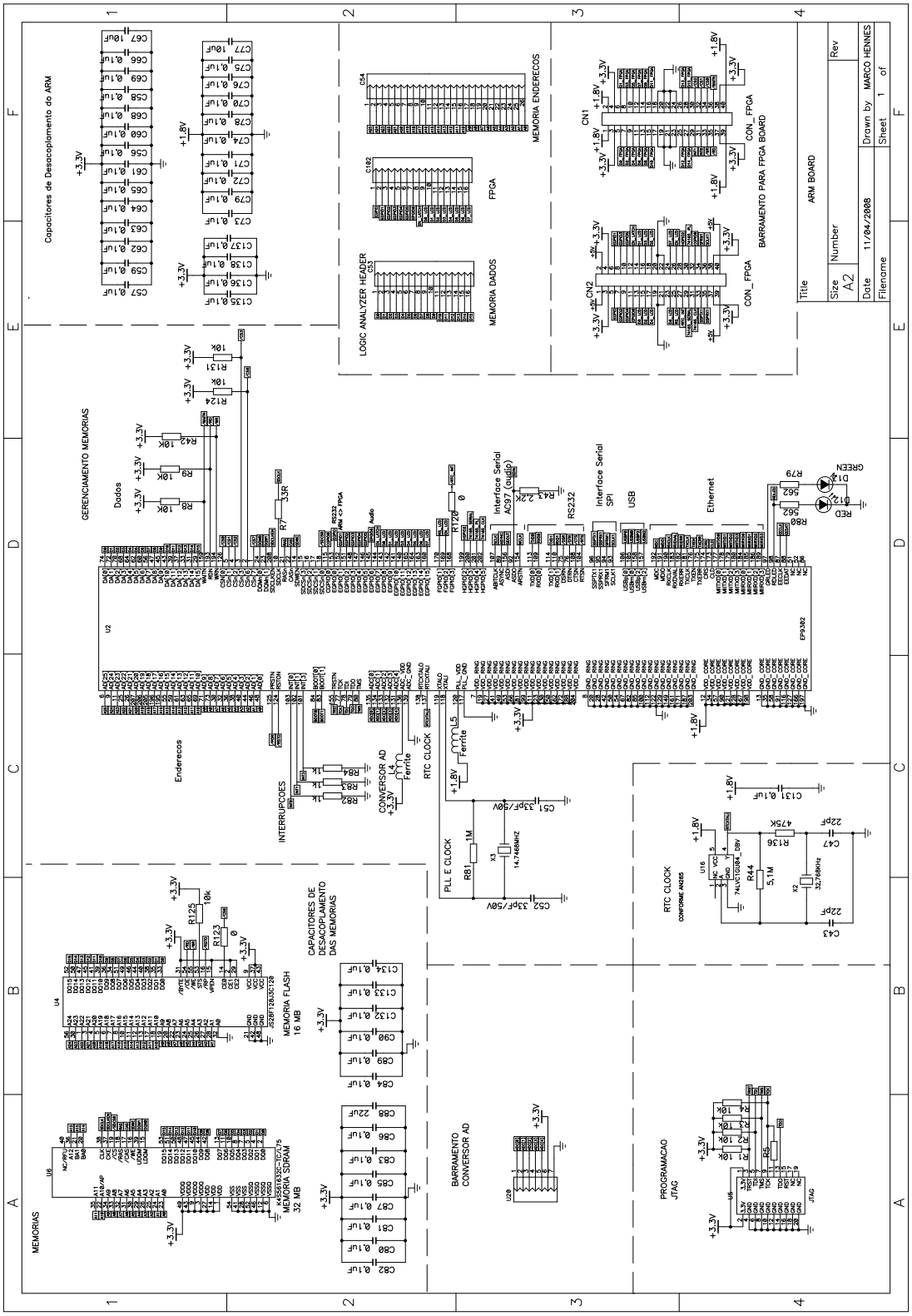


Figura 63: Placa de circuito impresso(1/4)

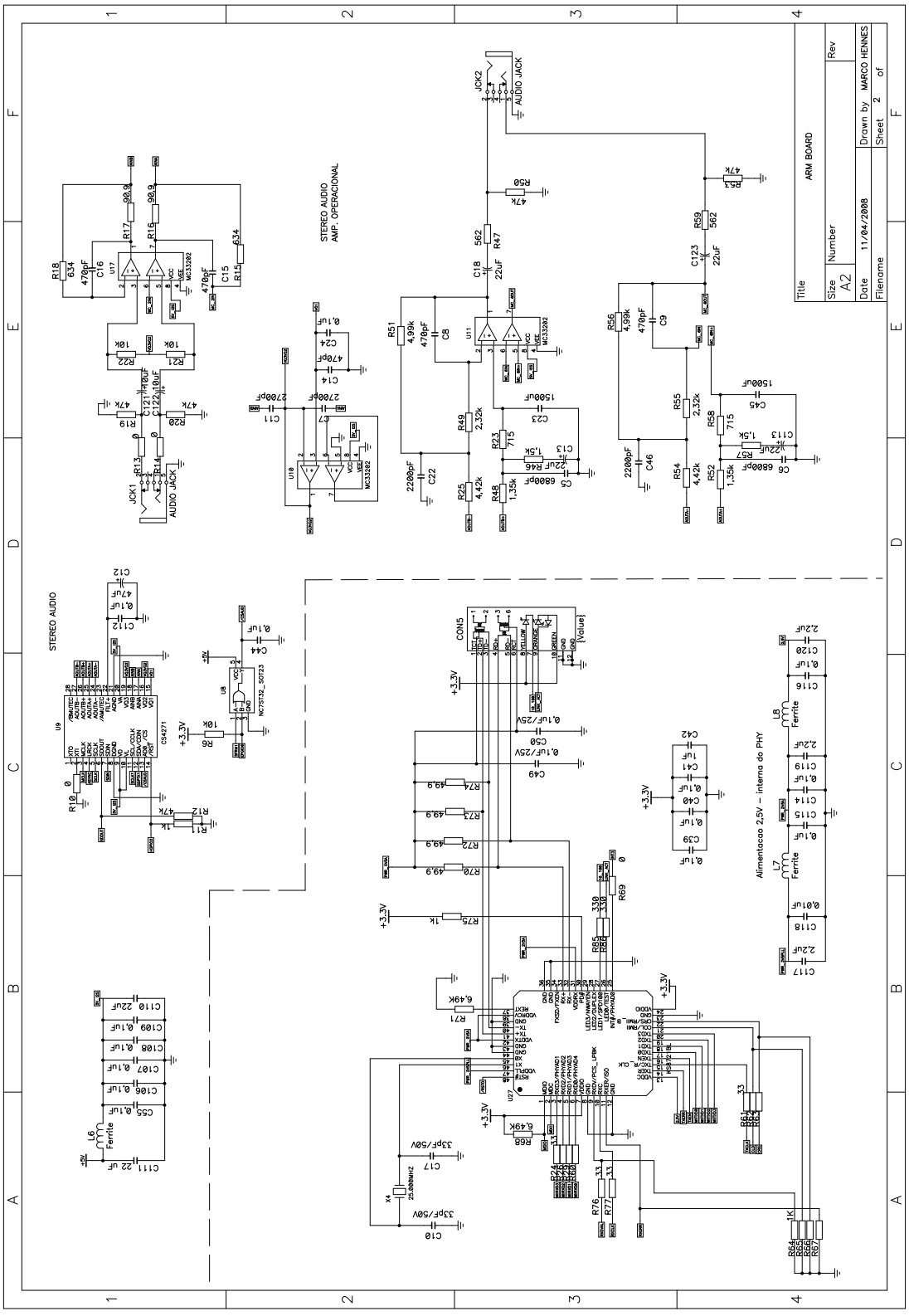


Figura 64: Placa de circuito impresso(2/4)

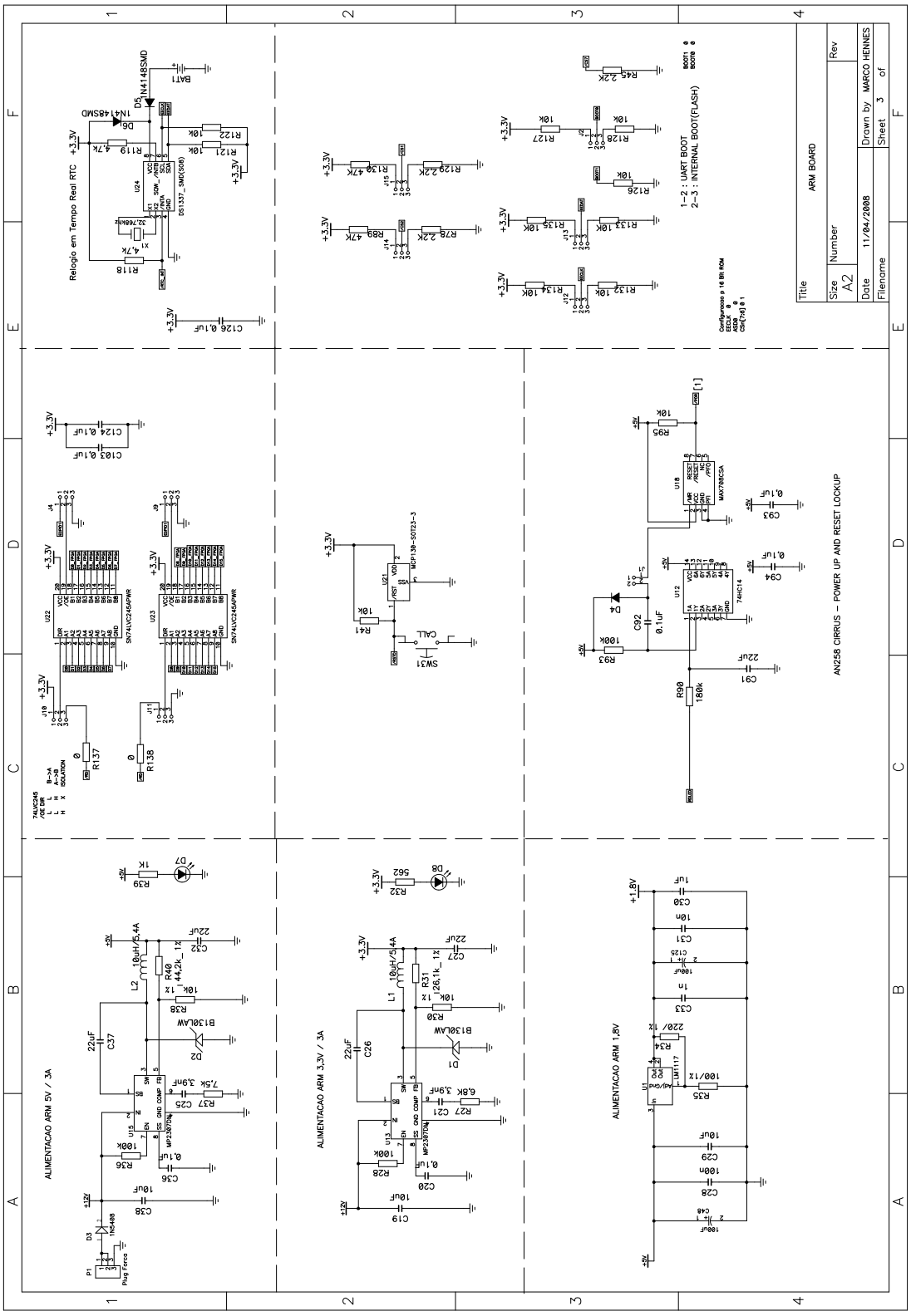


Figura 65: Placa de circuito impresso(3/4)

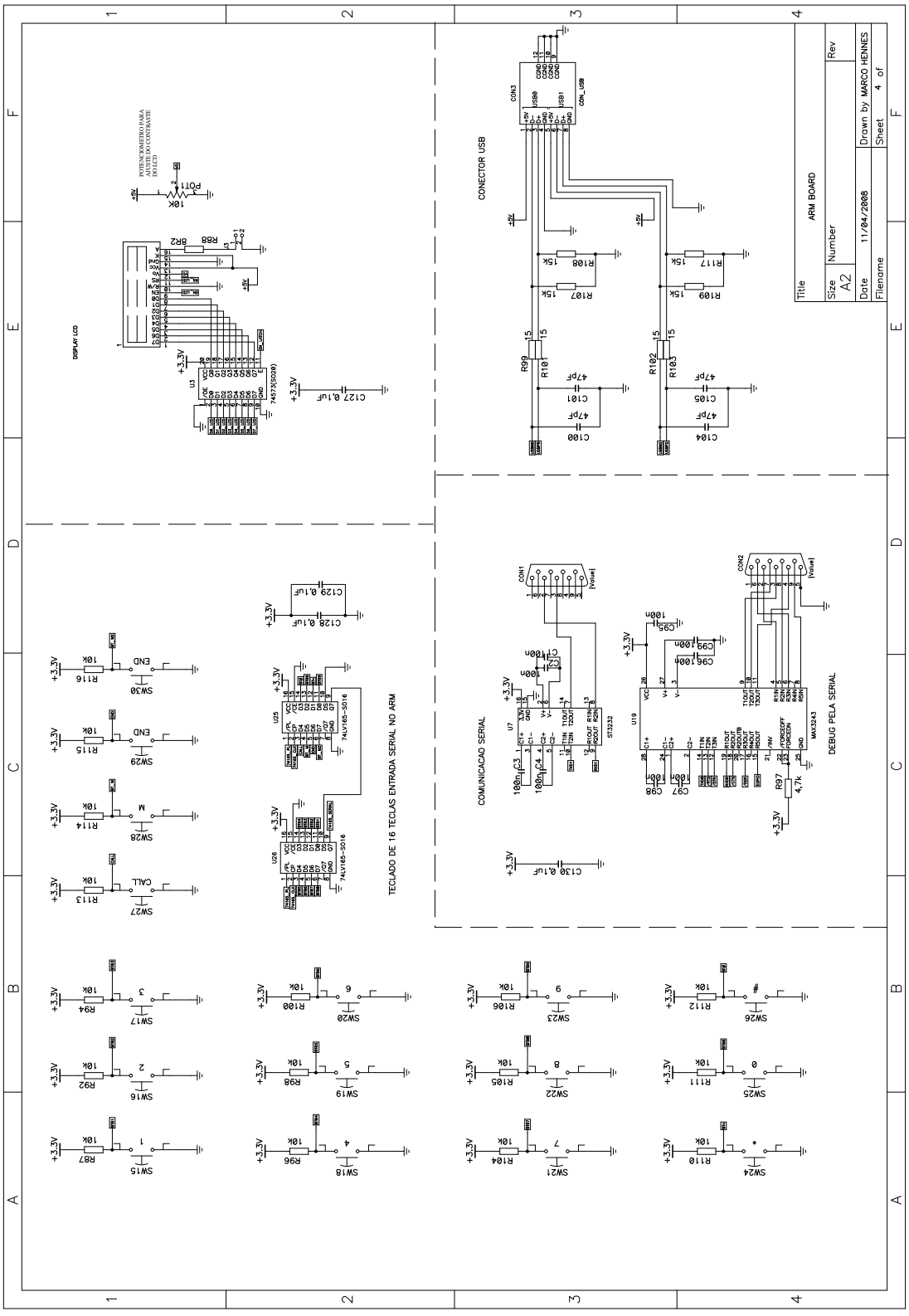


Figura 66: Placa de circuito impresso(4/4)

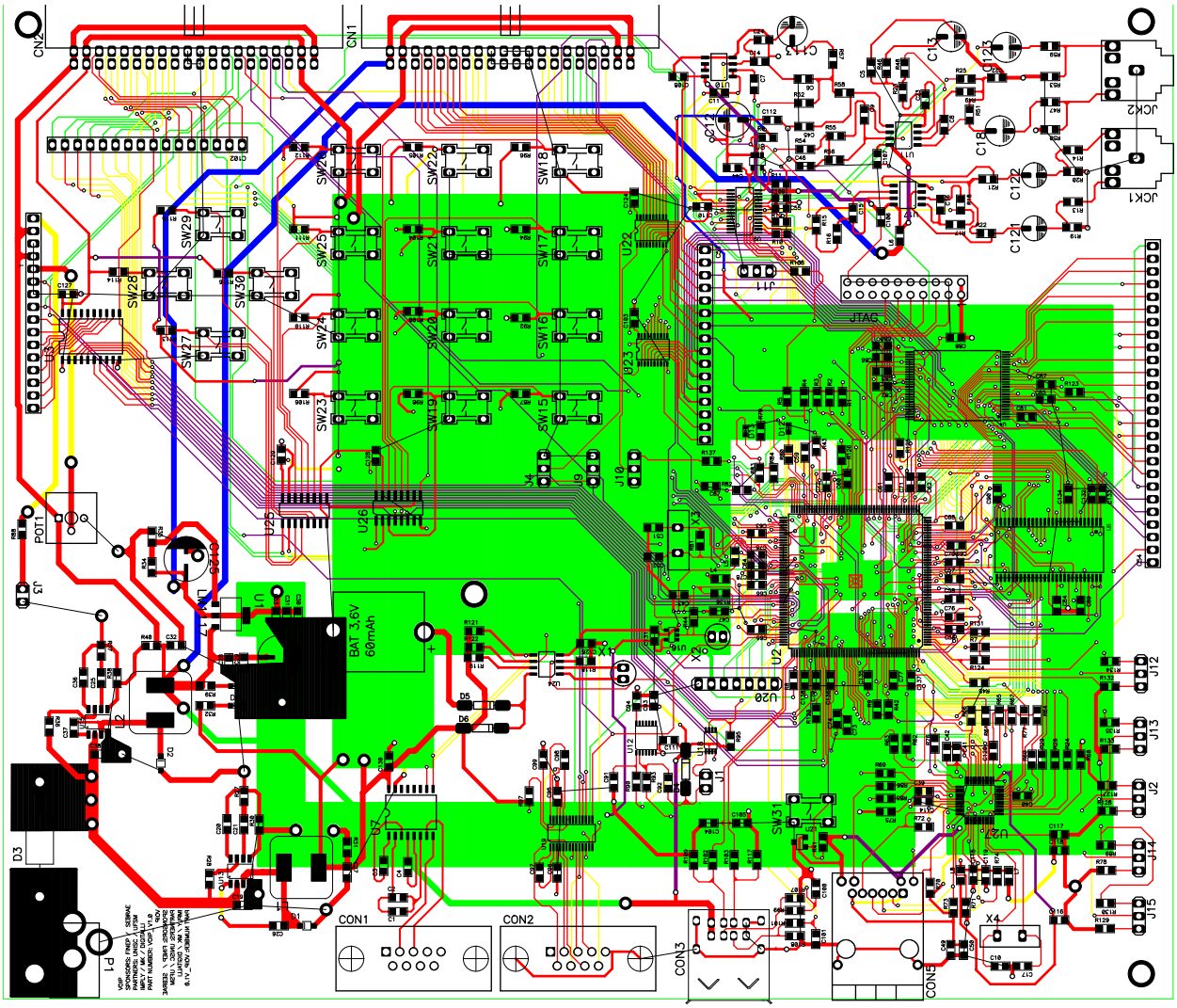


Figura 67: Roteamento da PCB